THE CORBA OBJECT GROUP SERVICE: A SERVICE APPROACH TO OBJECT GROUPS IN CORBA

THÈSE N^{O} 1867 (1998)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Pascal FELBER

Ingénieur informaticien diplômé EPF
originaire de Lausanne (VD)

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Prof. L. Moser, rapporteur
Dr J. Sventek, rapporteur
Prof. A. Wegmann, rapporteur

Lausanne, EPFL
1998

Abstract

Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures. To answer the growing demand in distributed technologies, several middleware environments have emerged during the last few years. These environments however lack support for "one-to-many" communication primitives; such primitives greatly simplify the development of several types of applications that have requirements for high availability, fault tolerance, parallel processing, or collaborative work.

One-to-many interactions can be provided by group communication. It manages groups of objects and provides primitives for sending messages to all members of a group, with various reliability and ordering guarantees. A group constitutes a logical addressing facility: messages can be issued to a group without having to know the number, identity, or location of individual members. The notion of group has proven to be very useful for providing *high availability* through *replication*: a set of replicas constitutes a group, but are viewed by clients as a single entity in the system.

This thesis aims at studying and proposing solutions to the problem of object group support in object-based middleware environments. It surveys and evaluates different approaches to this problem. Based on this evaluation, we propose a system model and an open architecture to add support for object groups to the CORBA middleware environment. In doing so, we provide the application developer with powerful group primitives in the context of a *standard* object-based environment. This thesis contributes to ongoing standardization efforts that aim to support fault tolerance in CORBA, using entity redundancy.

The group architecture proposed in this thesis — the *Object Group Service* (OGS) — is based on the concept of *component integration*. It consists of several distinct components that provide various facilities for reliable distributed computing and that are reusable in isolation. Group support is ultimately provided by combining these components. OGS defines an object-oriented framework of CORBA components for reliable distributed systems.

The OGS components include a *group membership service*, which keeps track of the composition of object groups, a *group multicast service*, which provides delivery of messages to all group members, a *consensus service*, which allows several CORBA objects to resolve distributed agreement problems, and a *monitoring service*, which provides distributed failure detection mechanisms. OGS includes support for dy-

namic group membership and for group multicast with various reliability and ordering guarantees. It defines interfaces for active and primary-backup replication. In addition, OGS proposes several execution styles and various levels of transparency.

A prototype implementation of OGS has been realized in the context of this thesis. This implementation is available for two commercial ORBs (Orbix and VisiBroker). It relies solely on the CORBA specification, and is thus portable to any compliant ORB. Although the main theme of this thesis deals with system architecture, we have developed some original algorithms to implement group support in OGS. We analyze these algorithms and implementation choices in this dissertation, and we evaluate them in terms of efficiency. We also illustrate the use of OGS through example applications.

Résumé

Les systèmes distribués occupent une place de plus en plus importante dans l'informatique actuelle. La distribution rend les systèmes plus complexes à concevoir et à gérer car ces derniers doivent tenir compte de nouveaux types de problèmes, tels que les pannes partielles ou les défaillances du réseau. Afin de répondre à la demande grandissante des technologies distribuées, plusieurs environnements de type "middleware" sont apparus au cours des dernières années. Ces environnements ne disposent cependant pas de primitives de communication permettant d'envoyer un message à plusieurs destinataires *(un vers plusieurs)*; ces primitives simplifient grandement le développement de plusieurs types d'applications qui ont des besoins spécifiques dans des domaines tels que la haute disponibilité, la tolérance aux défaillances, le calcul parallèle ou le travail collaboratif.

La communication de groupe offre un outil adéquat pour les interactions de type *un vers plusieurs*. Elle gère des groupes d'objets et permet d'envoyer des messages à tous les members d'un groupe avec diverses garanties de fiabilité et d'ordonnancement. Un groupe représente une facilité d'adressage logique: des messages peuvent être envoyés à un groupe sans avoir à connaître le nombre, l'identité ou l'emplacement de chacun de ses membres. La notion de groupe a prouvé son utilité dans le contexte de la haute disponibilité par duplication: plusieurs copies d'un objet sont réunies dans un groupe qui apparaît à ses clients comme une seule entité dans le système.

Le but de cette thèse est d'étudier et de proposer des solutions au problème de la communication de groupe dans un environnement orienté-objet de type "middleware". Elle présente et évalue différentes approches à ce problème. Sur la base de cette évaluation, nous proposons un modèle et une architecture ouverte permettant d'ajouter la notion de groupe d'objets dans l'environnement CORBA. Par cela, nous mettons à disposition du programmeur un ensemble de primitives puissantes pour la communication de groupe dans le cadre d'un environnement orienté-objet *standard*. Cette thèse contribue aux efforts actuellement en cours visant à supporter la tolérance aux défaillances dans CORBA à l'aide de la redondance.

L'architecture proposée dans cette thèse — le service de groupes d'objets (OGS) — est basée sur le concept d'*intégration de composants*. Cette architecture comprend différents composants, réutilisables indépendamment les uns des autres, qui mettent à disposition diverses facilités pour la programmation d'applications distribuées fiables. La communication de groupe est mise en oeuvre à l'aide de ces différents composants. OGS définit donc un ensemble de composants CORBA pour

les systèmes distribués fiables.

Les composants d'OGS incluent un service qui gère la composition de groupes d'objets, un service qui fournit diverses primitives pour communiquer avec ces groupes, un service qui permet de résoudre le problème du consensus distribué, et un service qui détecte les pannes de composants distants. OGS gère des groupes dont la composition peut évoluer dynamiquement, et offre diverses garanties de fiabilité et d'ordonnancement pour la communication de groupe. Il propose des interfaces permettant de dupliquer des objets activement ou passivement. En outre, OGS offre différents styles d'exécution et divers niveaux de transparence.

Un prototype d'OGS a été mis en oeuvre dans le contexte de cette thèse. Ce prototype est disponible pour deux mises en oeuvre commerciales de CORBA (Orbix et VisiBroker). Comme il se base uniquement sur le standard CORBA, ce prototype peut être facilement porté sur toute autre mise en oeuvre de CORBA. Bien que la contribution principale de cette thèse se situe au niveau architectural, nous avons adopté des approches algorithmiques originales pour la communication de groupe. Nous détaillons ces algorithmes dans cette thèse et nous évaluons leurs performances. En outre, nous illustrons l'utilisation d'OGS à l'aide de plusieurs exemples d'applications.

To Aline, Agnès, and
all of you who read this...

Acknowledgments

I wish to express my gratitude to my supervisor, Prof. André Schiper, for his continuous support during the four years that I have spent in his Laboratory. His confidence in me, his sense of clarity and precision, his patience and open-mindedness have been essential factors to the success of this research.

My thanks go to the members of the jury for the time they have spent applying their expert knowledge to the examination of this thesis: Prof. Louise Moser, Dr Joe Sventek, and Prof. Alain Wegmann. I am also grateful to Prof. Boi Faltings for having presided over the jury, and to Prof. Michael Melliar-Smith for having attended to the examination.

I wish to thank all the people who have helped me in realizing my ideas. I have greatly appreciated working with all my colleagues at the Operating Systems Laboratory; they have been very helpful throughout the years that we have spent together. I am particularly grateful to Kristine Verhamme for her constant help with all the administrative problems that I would have been unable to solve myself. I am also thankful to Bernhard Ruch and Patrick Eugster who have contributed to early implementations of OGS.

Special thanks go to Dr Rachid Guerraoui for his constant encouragement, his great help in clarifying and synthesizing the results of my work, and the fruitful collaboration we had throughout the OpenDREAMS project.

I also wish to express my gratitude to Xavier Défago, Patrick Eugster, Rachid Guerraoui, David Nathan, and André Schiper, who have spent time reading and commenting on early versions of this dissertation.

Finally, I wish to thank my family, my friends, and my wife Aline for their faithful and loving support.

Contents

List of Figures

List of Tables

Introduction

It is not certain that everything is uncertain.

B. Pascal

Context

New challenges for software developers are the result of growing interest in *distributed* technologies, driven primarily by the popularization of the Internet and the now ubiquitous *World Wide Web* (WWW). Distributed applications must deal with complex issues, such as remote communication, partial failures, distributed garbage collection, and concurrency management.

The last few years have seen the emergence of several programming environments that greatly reduce the complexity of developing distributed software. These environments, regrouped under the term *middleware* because they appear between application programs and operating system services, provide high-level facilities for developing distributed applications without having to deal with low-level details, such as remote communication and object location. They use object-oriented concepts to abstract the complexity of the system and promote modularity and reusability. These environments offer frameworks for integration of heterogeneous distributed components. Examples of these middleware architectures are OMG's CORBA [OMG98a] and Microsoft's DCOM [Ses97].

Motivations

Existing object-oriented middleware environments essentially deal with point-to-point invocations. While this interaction style complies with the invocation model of object-based systems, some types of applications need to invoke *several objects* at once. In particular, in the contexts of fault tolerance, load balancing, or parallel processing, "one-to-many" communication facilities have been shown to be useful.

One-to-many interactions can be provided by group communication. Group communication manages groups of objects and provides primitives for sending messages to all members of a group, with various reliability and ordering guarantees. A group constitutes a logical addressing facility: messages can be issued to a group without having to know the number, identity, or location of individual members. The notion of group has proven to be very useful for providing *high availability* through *replication*: a set of replicas constitutes a group, but are viewed by clients as a single entity in the system.

Currently, systems that require object group support must do all the necessary design and implementation related to group communication, with correspondingly no guarantee of either interoperability or portability. The aim of this thesis is to study and propose a new approach for integrating group support in the CORBA middleware architecture, which complies with existing standards. CORBA does not currently provide any support for object groups. Providing group support in CORBA will reduce the burden on designers and implementers of fault tolerant applications. Applications will benefit from the power of groups (high availability, fault tolerance, etc.) while preserving the key features of object-oriented middleware environments (simple development process, distribution transparency, component integration, etc.).

The relevance of this thesis is emphasized by the fact that, at the time of writing, the OMG had issued a *Request For Proposal* (RFP) for fault tolerance support in CORBA using entity redundancy. A further motivation of this work is to offer valuable contributions to this RFP.

Contributions

Groups in Object-Based Distributed Systems. This thesis starts by presenting surveys of object-based and group-based distributed systems. Although object-oriented middleware offers many advantages when developing and deploying component-based distributed applications, we argue that its underlying point-to-point invocation model limits its suitability for various types of applications. We outline the requirements of such applications regarding group support, which provides an adequate paradigm for fault tolerant and highly available distributed computing. We present and evaluate different approaches followed by existing systems to support groups in the CORBA middleware environment, and we argue that they are not fully consistent with the modular, component-based architecture promoted by CORBA. These approaches can be classified according to three categories: the *integration* approach, the *interception* approach, and the *service* approach.

A Service Approach. We introduce a service-based approach to support object groups in CORBA, that provides group support as an optional CORBA component. Application may use groups together with other CORBA functionalities through *component integration*. By adopting a service approach, we inherit from the major CORBA features, such as heterogeneity, portability, interoperability, modularity,

and reusability. We propose several models of transparency, and original solutions to tackle the problems related to group management in object-oriented architectures. We present portable and interoperable tools for encapsulating group behavior and plurality, and for hiding object failures from clients.

We highlight our approach together with the ongoing efforts of the OMG, which aims at supporting fault tolerance in CORBA using entity redundancy. We believe that the contribution of this thesis and the lessons learned from our experiences can be useful for this ongoing specification.

The Object Group Service Architecture. We propose the design of a CORBA *Object Group Service* (OGS), that provides group support for standard off-the-shelf CORBA environments. The OGS environment specifies an architecture and a set of interfaces for object groups. The OGS architecture does not define a single monolithic component; it is decomposed into several CORBA services that provide various facilities for reliable distributed computing, and that are used for the actual implementation of group communication. In particular, the *Object Monitoring Service* provides distributed failure detection mechanisms, and the *Object Consensus Service* allows several CORBA objects to solve distributed agreement problems. This decomposition into several independent components promotes modularity and reusability, and extends the results of this thesis to areas other than group communication.

In contrast with other systems that use group communication, OGS does not limit itself to replication, but extends the use of object groups to other types of applications, such as parallel processing, load sharing, and cooperative work.

A Prototype Implementation. The OGS architecture has been implemented as a prototype that acts as a proof of concept. Portability has been demonstrated by compiling our implementation with two commercial CORBA environments (Orbix and VisiBroker). In addition, we have developed one implementation in C++ and one in Java that interoperate with each other. We have been able to provide a high level of transparency, while using only constructs defined in the CORBA specification.

Although the main theme of this thesis deals with system architecture, we have developed some original algorithms to implement group support in OGS. We analyze these algorithms and implementation choices in this dissertation, and we evaluate them in terms of efficiency. We also illustrate the use of OGS through example applications.

Roadmap

This dissertation is organized as follows: Chapter 1 presents background concepts about distributed computing, group communication, object-based systems, middleware environments, and CORBA. We discuss the benefits of these paradigms, and

present a survey of existing systems based on them.

Chapter 2 introduces the main issue addressed by this thesis, i.e., group support in object-oriented distributed environments. We present basic notions related to the use of groups in object-based systems, and the problems that arise from this union. We introduce generic solutions to integrate both concepts, and give an architectural overview of the *Object Group Service* (OGS) developed in the context of this thesis.

Chapter 3 details the various components of OGS. Each component is a modular unit that provides generic services useful for many kinds of applications. Each component is architecturally independent of the other components. We describe the design and architecture of each component, its interfaces, its semantics, and how one can use its services. This chapter uses a top-down approach, presenting the components that are close to the application first.

Chapter 4 describes how the various components of OGS were developed and how they relate to each other in the current OGS implementation. We describe the distributed protocol model of OGS and the algorithms used to provide group communication. Although object-oriented decomposition offers many advantages, it also has various costs. These are evaluated and discussed.

Chapter 5 presents the different OGS execution models, and how it may be configured for several levels of reliability, transparency, and performance. We detail how OGS can be used with different programming languages, and we present several application examples that use OGS for various tasks.

The concluding chapter summarizes the major results of this thesis. We present how this work relates to the ongoing standardization efforts for object group support in CORBA, and we discuss the future of OGS.

Chapter 1

Background

I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated.

P. Anderson

1.1 Distributed Computing

1.1.1 A Brief History

Computing systems have evolved from centralized architectures to distributed systems. Distributed systems have evolved then from simple client/server applications running on *Local Area Networks* (LANs) to complex systems involving a huge number of machines across *Wide Area Networks* (WANs). This section presents a brief history of distributed computing [Bir96].

Network Computing Systems

A *networked application* is a computer application that consists of several decoupled components communicating by exchanging messages. The development of client/server networked architectures peaked during the 1980s, when it became possible to put the power of a mainframe on a desktop computer. The concerns of network computing are generally described in terms of the *Open Systems Interconnection* (OSI) layering. In this descriptive structure, several software layers abstract the details of the physical network, packet and message transmission, routing, data representation, addressing, and session management. Each layer is built using the services provided by the underlying layers. The *TCP/IP* protocol suite is a typical example of a network architecture that is closely matched with the OSI model. A

property inherent to the OSI model is that communication is limited to point-to-point data transmission.

Remote Procedure Calls

The next big evolution of distributed computing occurred with the introduction of *Remote Procedure Calls* (RPCs) [BN84]. RPCs allow client programs to *transparently* issue calls to procedures defined by remote server programs. The complexity of making connections and marshaling data in and out of messages is completely hidden from the application by *stubs* that mimic the interface of the procedure calls. Network operating systems have been hugely successful over the last 15 years, and RPC mechanisms have been extensively used in these operating systems for distributed services such as network file systems, name services, and synchronized clock services.

Distributed Computing Systems

In the late 1980s, with the availability of powerful desktop computers that can be interconnected through very fast networks, centralized multi-processor parallel architectures have been progressively replaced by distributed system architectures. The term *distributed computing*, in contrast with network computing, designates a set of *tightly coupled* programs executing on one or more computers and coordinating their actions. These programs know about one another and cooperate to perform a task that none could carry out in isolation. Such systems allow the sharing of information and resources, and may be composed of small, cost-effective computers that combine their processing power.

A typical example of a distributed computing system is the *Parallel Virtual Machine* (PVM) [Sun90], which is a software package that permits a heterogeneous collection of computers hooked together by a network to be used as a single large parallel computer. Thus large computational problems can be solved at low cost by *temporarily* using the combined power and memory of many computers.

While distributed computing is appealing, since it allows us to decompose and extend applications in a very flexible and powerful way, it is harder to manage because we have to address issues such as independent failures, unreliable communication, and insecure communication.

1.1.2 Reliable Systems

Since more things can go wrong in distributed systems, they are intrinsically less reliable than centralized systems. Many mechanisms have to be built in, e.g., to handle the fact that the client and the server may fail independently *(partial failures)*. The network necessary for remote communication between distributed components is another source of unreliability that distributed applications have to deal with. Some well-known types of reliability [Bir96] are:

- **Fault tolerance:** fault tolerance is the ability of a distributed computing system to recover from the failure of some component. A component is considered *faulty* once its behavior is no longer consistent with its specification.
- **High availability:** a highly available system provides uninterrupted service in spite of failures.
- **Consistency:** consistency is the ability of a distributed computing system to coordinate related actions of multiple components, despite concurrency and failures. It generally encompasses the ability of making a distributed system behave like a non-distributed system.
- **Security:** security is the ability of a system to protect data, services, and resources against unauthorized access.
- **Privacy:** privacy is the ability of a system to protect user identity and data from other users.

For the purpose of this dissertation, we focus on fault tolerance, availability, and consistency, although the model and some of the solutions proposed may be applied to other aspects of reliability. Tolerating failures requires a clear definition and understanding of failures. Some types of failures that may occur in distributed systems [Bir96] are:

- **Halting failures:** in the halting failures model, a process either works correctly, or simply stops and crashes without performing incorrect actions.
- **Fail-stop failures:** in the fail-stop failures model [SS83], in addition to the process crashing without performing incorrect actions, processes that are interacting with the faulty process have an accurate way to detect such failures.
- **Send-omission failures:** these are failures to send a message that should have been sent.
- **Receive-omission failures:** these are failures to receive a message that has actually been sent and correctly transmitted by the communication channel. This may happen because of a lack of memory for buffering messages in the destination process.
- **Network failures:** these are failures that occur when the network loses messages, or is fragmented in disconnected subnetworks that cannot communicate with each other.
- **Timing failures:** these failures appear when a temporal property of the system is violated.
- **Byzantine failures:** Byzantine failures [LSP82] encompass a wide variety of faulty behavior. Malfunctioning or malicious processes can send messages with wrong data, or omit sending messages when they have to. These types of failures are difficult to even detect, let alone correct, since failed processes

exhibit unpredictable behavior; some systems, such as Rampart [Rei95], focus on this problem.

Of course, combinations of these unwanted behaviors may also appear. In this work, we only consider the halting failure model. We assume that the only type of process failure is a *crash*, in which the process simply halts, losing its volatile data. But the general architecture presented in this dissertation can adapt to other system models.

1.2 Group-Based Computing

1.2.1 Group Communication

Groups were first introduced in the V-Kernel [CZ85], as a convenient addressing mechanism. They were later extended to handle replication in the Isis system [Bir93]. The key idea of *group communication* is to gather a set of processes or objects into a logical group, and to provide primitives for sending messages to all group members at the same time with various ordering guarantees. A group constitutes a logical addressing facility since messages can be issued to groups without having to know the number, identity, or location of individual members. Groups have proven to be very useful for providing *high availability* through *replication*: a set of replicas constitutes a group, viewed by clients as a single entity in the system.

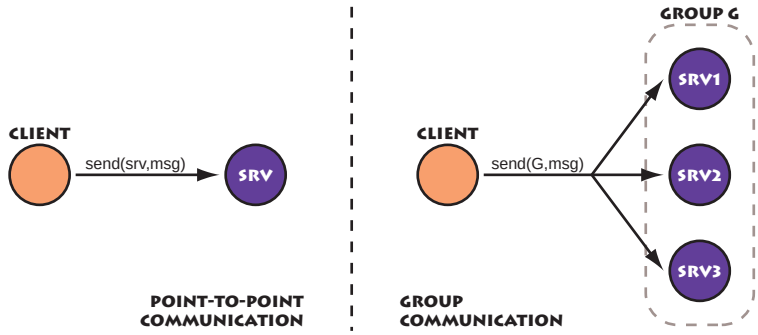

Figure 1.1: Point-to-Point vs. Group Communication

Figure 1.1 illustrates how messages are sent from a client to a server using point-to-point communication, and to a group of servers using group communication. Group communication toolkits generally provide message-passing interfaces that are very similar to those of standard point-to-point messaging libraries, but expect a group identifier as destination instead of a remote address. The client issues a single function call, which leads to the diffusion of the message to all group members.

Groups may be *static* or *dynamic*. A static group is a group whose membership does not change during the system's lifetime. Members that crash are not excluded from the group. Dynamic groups are groups whose membership changes over time, as the result of the crash of a member, or of a new member (re-)joining the group.

Dynamic groups are more powerful than static ones, but they are also more complex to implement.

Groups may be modeled as closed structures, where only members can issue multicasts to their group, or as open structures, where non-members can issue multicasts as well.

Total Order

Total order multicast is one of the most useful primitives for group communication. Simply stated, it ensures that messages sent to a group are delivered in the *same* order to *all* members of the group. Total ordering of messages is required for instance in replication, to ensure that the replicated data is kept consistent.

Figure 1.2 shows two messages sent by two clients to a set of servers with and without total ordering. A total order protocol may need to delay the actual delivery of a message to the application for ensuring correct ordering.

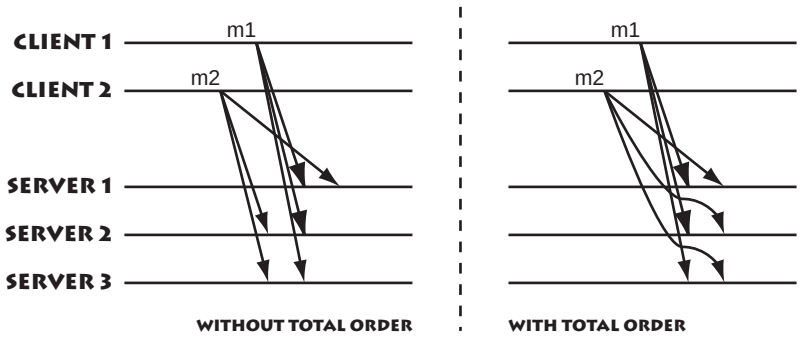

Figure 1.2: Total Order Multicast

Virtual Synchrony and View Synchrony

In asynchronous systems, no assumption is made about the transmission delay of messages, nor about the relative speed of processes in the network. The *virtually synchronous* execution model, first introduced by Birman [BJ87], provides the simple abstraction of a set of processes or objects — the group members — which all see the same events,[1] in the same order. Events are synchronous in terms of logical time, and asynchronous in terms of physical time. Since all group members see the same inputs, they can execute the same algorithm and have consistent states, assuming their behavior is deterministic.

A key element of the virtually synchronous execution model is that all members of a group are presented with identical sequences of group membership, called *views*, with mutually consistent rankings of group members.

[1] Events are incoming messages and group membership changes.

Unlike virtual synchrony, the *view synchronous* execution model does not include causal and total ordering of messages. This work is based on a view synchronous execution model in which total order also guarantees that messages are ordered with respect to view changes.

Figure 1.3 illustrates how view synchrony affects the respective ordering of messages and views upon the failure of a group member. With view synchrony, messages are delivered *before* or *after* the view change by all group members.

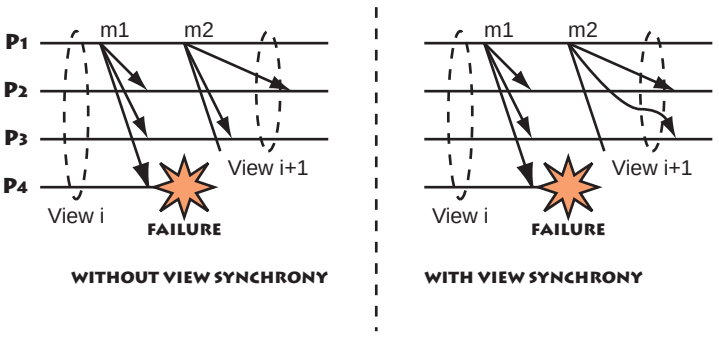

Figure 1.3: View Synchrony

State Transfer

Since the members of a group usually share a common state, a new member has to receive the current state from the group when dynamically joining it. This is performed by a *state transfer* mechanism that transmits the state from a current member of the group to the new one. The state transfer is generally driven by the group communication system that calls back to the application for accessing the state. Hence, group members must provide operations for "getting" and "setting" their state.

1.2.2 Replication

The idea of using redundancy as a mean of masking the failures of individual components dates back to von Neumann [vN56]. With redundant copies, a replicated entity can continue providing services in spite of the failure of some of the copies, without affecting its clients. Redundancy may appear at different points in the architecture, such as redundancy of computational and storage resources; redundancy of communication links between these resources and their clients; redundancy of transient application components.

In distributed systems, the two best known replication policies are *active* and *primary-backup* replication. A replicated object is represented by a set of copies. This set may be static or dynamic. Static replication requires that the number and the identity of the copies do not change during the lifetime of the replicated object. Dynamic replication is more powerful since copies may be added or removed at runtime.

Active Replication

Active replication — also called the *state machine approach* — is a general protocol for replication management that has no centralized control [Sch93a]. All copies of the replicated object play the same role: they all receive each request, process it, update their state, and send a response back to the client (Figure 1.4). Because of the invocations always being sent to every replica, the failure of one of them is transparent to the client.

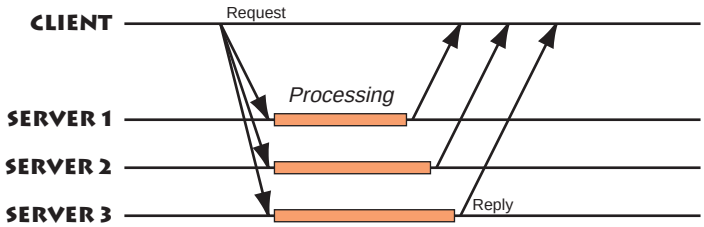

Figure 1.4: Active Replication

Active replication requires the operations on the replicated object to be deterministic. Determinism means that the outcome of an operation depends only on the initial state of the object, and on the sequence of operations performed by the object (history). If the operations on the replicated object are deterministic, the shared state of the replicated object remains consistent and all responses sent back to the client are identical. The client typically waits for the first reply, making the failure of a server transparent.

To a client, all correct replicas should appear as having the same state. In order to guarantee this, all invocations sent by the clients should be treated in the same order by all correct replicas. This is ensured by a *total order multicast* primitive [HT93] — also called *atomic multicast* — that provides total ordering of messages multicast to a set of destinations.

Primary-Backup Replication

With *primary-backup replication* — also called *passive replication* — one server is designated as the *primary*, while all other are *backups* [BMST93]. Clients perform requests by sending messages only to the primary, which executes the request, and atomically updates the other copies and sends the response to the client upon completion (Figure 1.5). If the primary fails, then one of the backups takes over. This scheme ensures linearizability[2] because the order in which the primary receives invocations defines the order for all servers. The acknowledgement sent by the backups and awaited by the primary ensures request atomicity.

Unlike active replication, primary-backup replication does not waste extra resources

[2]Linearizability guarantees that each operation invoked on a server is performed on the latest state of the server [HW90].

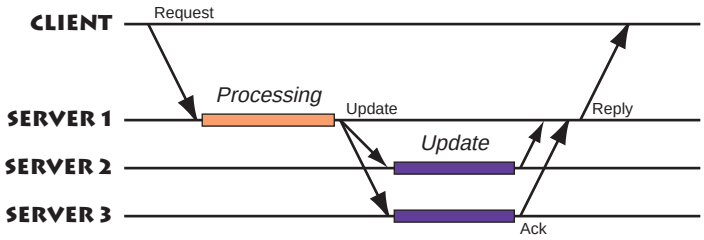

Figure 1.5: Primary-Backup Replication

through redundant processing, and permits non-deterministic operations. However, responses will be delayed by the failure of the primary. Furthermore, primary-backup replication requires additional application support for the primary to update the state of the other copies.

1.2.3 Group-Based Systems

Several systems provide support for managing groups in distributed environments. Most of these systems offer procedural tools for group communication and consider process groups, in contrast with object groups. We give a short overview of the major group communication systems, which are relevant in the context of this thesis.

Isis

The Isis Toolkit [BV93], developed at Cornell University, was the first group communication system to support virtual synchrony. Isis is a collection of procedural tools that are linked directly to the application program, providing it with the functionality for creating and joining process groups dynamically, and multicasting messages to process groups with various ordering guarantees.

Isis provides virtual synchrony with FIFO, causal, and total order multicast primitives. The membership protocol of Isis provides primary partition semantics, i.e., in case of a network partition there is always only one set of processes, located in the primary partition, that can process requests on behalf of the group. Isis does not provide adequate support for group-to-group communication (i.e., invocations from one group to another group).

Horus

The Horus project [vRBM96] was originally launched as an effort to redesign the Isis group communication system in a set of small, clearly defined modular units. Within the Horus framework, a large collection of system and application protocols have been developed that allow the application designer to construct a communication module that exactly meets the application's requirements at a minimal cost.

Horus provides a virtual synchrony model that lets an application make progress even in minority partitions, indicating to the application whether it is currently part of the primary partition or not. Horus also provides support for the development of distributed systems in situations where Isis is unsuitable, such as applications that have special real-time requirements.

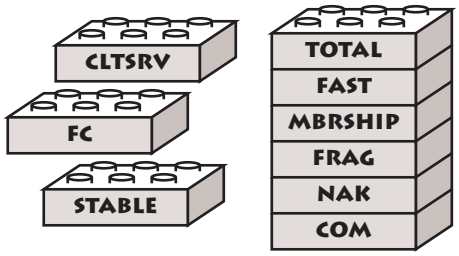

Figure 1.6: Horus Protocol Layers can be stacked like Lego^{TM} Blocks.

One of the most interesting features of Horus lies in its communication architecture that treats a protocol as an abstract data type. Protocol layers can be stacked on top of each other in a variety of ways at runtime, as shown in Figure 1.6. Horus provides a *protocol composition* framework [vRBF+95], that allows custom protocols to be built from composing existing ones; this architecture has the additional advantage that an application only pays for the properties it uses.

Ensemble [Hay98] is the next generation of the Horus group communication toolkit, written in the ML programming language.

Totem

The Totem system [MMSA+96], developed at the University of California, Santa Barbara, is a set of communication protocols for the construction of fault tolerant distributed systems. Totem provides several ordered multicast primitives to process groups, with high throughput and low, predictable latency. These protocols exploit the hardware multicast capabilities of local-area networks and the locality of process groups in order to provide soft real-time guarantees.

Totem provides reliable total order primitives within process groups in a LAN, using protocols based on a logical token-passing ring [AMMS+95]. The Totem system handles processor failure and recovery, network partitioning and remerging (with continued operation of all parts of a partitioned system). It provides a membership service adapted to the extended virtual synchrony model, which is a virtual synchrony model with minority partitions (also adopted in Horus and Transis). Totem also provides reliable total order primitives within process groups in multiple LANs interconnected by gateways to achieve greater scalability and lower latency than a single ring can provide [AMMSB98].

Transis

The Transis communication system [ADKM92], developed at the Hebrew University of Jerusalem, supports process group communication with several forms of group multicast operations: FIFO ordered, causally ordered, totally ordered, and safely delivered.

Transis contains a protocol for reliable message delivery that optimizes the performance for existing network hardware and tolerates network partitioning. It employs an efficient multicast protocol, based on hardware multicast. The Transis system is based on the *Trans* protocol developed at UCSB [MSMA90, MMSA94]. Transis also supports partitionable operating, and provides the means for consistently merging components upon recovery.

Phoenix

Phoenix [Mal96], developed at the Swiss Federal Institute of Technology, Lausanne, is a group communication toolkit that provides view synchrony in asynchronous large scale environments (wide geographical distribution and high number of participating processes).

Phoenix implements dynamic routing, datagram-oriented reliable communication channels, group communication primitives, and view synchrony. It supports a high number of processes by assigning them different roles: *core members*, *clients*, and *sinks*. Core members manage shared state and have the strongest reliability guarantees with respect to message delivery and membership changes. They communicate with each other using group multicasts. Clients interact with members by sending requests to them, and receiving replies and view changes. An interaction between a client and a member is more efficient than between two members, but the former offers weaker reliability guarantees than the latter. Finally, sinks only receive information diffused by the members of the group. As suggested by their name, sinks cannot perform requests and only receive messages. Figure 1.7 illustrates the different roles of Phoenix objects.

The Phoenix API [FG95] provides object-oriented abstractions for supporting object groups instead of process groups. Core members, clients, and sinks appear in a clean inheritance hierarchy. Core members inherit from the properties and behavior of clients, which inherit in turn from sinks. Phoenix objects can hence combine several roles.

1.3 Object-Based Computing

Booch [Boo94] presents the inherent complexity of software as deriving from four elements: the complexity of the problem domain, the difficulty of managing the developmental process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems. A common goal to most analysis and design methods is to hide this inherent complexity, and to give the illusion of

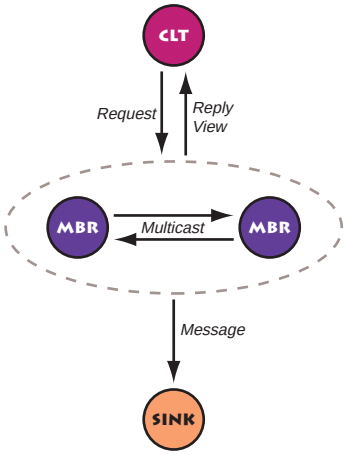

Figure 1.7: Phoenix Members, Clients, and Sinks.

simplicity. Dealing with complexity may be addressed through the use of *decomposition*, *abstraction*, and *hierarchy*. Object-oriented analysis and design is a method based on object-oriented decomposition where the complex problem is viewed as a meaningful collection of small objects that collaborate to achieve some higher level behavior.

1.3.1 Concepts

The major goal of *Object-Oriented Programming* (OOP) [Weg90] is to provide a better programming model for representing the world. The basis of the object model is the concept of *objects*, which are entities modeled on the real world, that have specific properties and exhibit specific behavior. An object is an instance of a *class*. Objects provide the means for combining behavior and state (i.e., program and data) into a single entity. The four major concepts underlying OOP are abstraction, encapsulation, inheritance, and polymorphism.

Abstraction

Abstraction is a key concept of object-oriented design. It is one of the fundamental ways to cope with complexity. An abstraction focuses on the outside view of an object and offers a simple and concise representation of a more complicated idea. The aim of abstraction is to hide complexity: complexity does not disappear, but does move to a more appropriate point in the architecture.

Encapsulation

Encapsulation hides the implementation details of an object from the services it can provide, by separating the contractual interface of an abstraction and its im-

plementation. Other objects can request services by sending messages to the object that provides the service. By decoupling the interface and the implementation, encapsulation enables us to modify an object's implementation without affecting its clients.

Inheritance

Inheritance enables us to pass along the capabilities and behavior of one class of objects to another. Inheritance defines a relationship between a *child* class that inherits from a *parent* class. The child class can be specialized by modifying inherited methods, or adding new ones. The parent is not affected by this modification. Inheritance enables us to *reuse* the properties of existing objects. Some systems support multiple inheritance, in which a child class may inherit from several parent classes.

Polymorphism

Polymorphism is the ability to substitute objects with matching interfaces for one another at runtime. With polymorphism, objects that have common descriptions can respond to the same request with different actions, depending on their type. Polymorphism makes it possible to design applications that are easily extensible.

1.3.2 Object-Oriented Frameworks

A framework is a reusable design expressed as a set of abstract classes and the way their instances collaborate [JF88]. This design is a solution for a family of problems. It describes how the software is decomposed into a set of interacting objects which have a given responsibility.

Frameworks are generally classified according to two different categories: *white-box* and *black-box* frameworks. White-box frameworks exhibit the internal structure of their class to the user. The application specific behavior is usually defined by adding methods to subclasses of the framework classes. Each method added to a subclass must abide by the internal conventions of its superclasses. Black-box frameworks are composed of components that provide the application-specific behavior. These components interact together using method invocations, so the user needs to understand only the external interface of the components.

Black-box frameworks are easier to use and learn than white-box frameworks, because the user is not required to have knowledge about internal details of the classes he uses. Black-box frameworks make it easy to change behavior at runtime, by replacing a component by another component with the same protocol. On the other hand, white-box frameworks are more flexible since they permit the definition of other behaviors than that supplied by the framework, and the number of possible combinations of components is not predetermined by the architecture of the

framework. In this thesis, we describe an architecture that defines a black-box style framework.

1.3.3 Objects in Distributed Computing

Several environments provide object-oriented abstractions and frameworks for distributed computing. Some of them mainly define *wrappers* on top of operating system services, such as the *Adaptative Communication Environment* (ACE) [Sch94]. Other infrastructures provide more elaborate facilities for distributed protocol composition, such as *Conduits+* [HJE95]. Some object-oriented programming languages include intrinsic support for distributed objects, such as Java *Remote Method Invocation* (RMI) mechanisms. Finally, some systems directly address reliability issues in distributed computing, and have similarities with the framework developed in the context of this thesis. This is the case of the GARF environment and the Bast framework, described below.

GARF

GARF [Maz96] is an environment aiming at promoting transparent distribution and replication of Smalltalk objects. GARF separates the distributed behaviors of objects from their functionalities, in order to simplify the programming of fault tolerant applications. Distributed objects are built from two distinct kinds of objects: *data objects*, that define their functionalities, and *behavioral* objects, that deal with distribution. While the former are application-specific, the latter are provided by the environment and transparently intercept all invocations sent or received by their associated data objects.

GARF defines two kind of behavioral objects: *encapsulators* and *mailers* [GGM93]. An encapsulator is associated with each data object, and a copy of the server's mailer resides on each node where a client is located. GARF provides a library of several ready-to-use behavioral objects that provide support for reliable distributed computing.

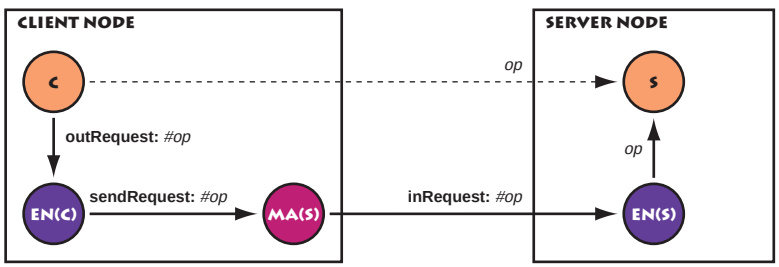

Figure 1.8: The GARF Invocation Model

Figure 1.8 illustrates the invocation mechanism of GARF, with a client C invoking the operation *op* of a server S. The invocation is intercepted by the encapsulator of C on the client node, and given to the mailer of S. This mailer performs a remote

invocation to the encapsulator of S, which invokes *op* on the server. The result comes back to the client through the reverse path. A remarkable characteristic of GARF is its symmetric invocation model [MGG95], which considers the request and the reply as instances of the same problem, and thus provides inherent support for group-to-group communication.

Bast

Bast [Gar98] provides a framework of protocol objects and patterns for structuring reliable distributed systems. It provides protocol composition facilities based on a white-box framework model. In Bast, reliability issues are modeled as elemental problem/solution pairs, thanks to *protocol objects* and *protocol patterns*. The former combine the features of distributed objects and those of protocols as first-class components, while the latter describe how to use protocol objects. Figure 1.9 gives an overview of the Bast architecture and its major components, which may be assembled to build protocol patterns. The Bast framework has been written in Smalltalk and ported to Java.

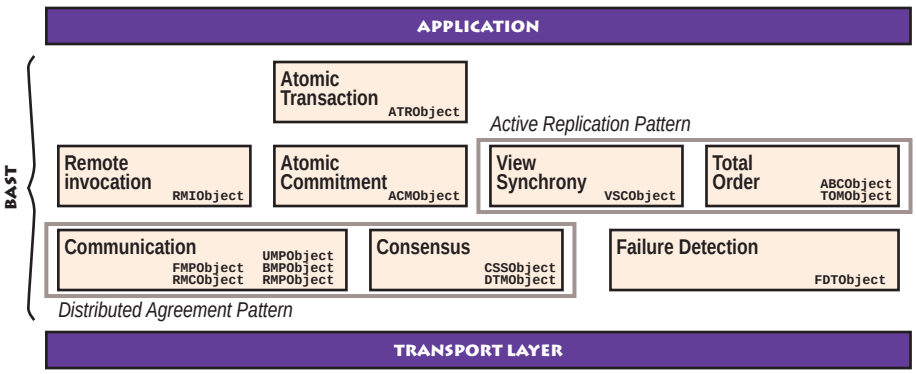

Figure 1.9: The Bast Architecture

1.4 The Common Object Request Broker Architecture

The *Common Object Request Broker Architecture* (CORBA) is an open, distributed, object computing infrastructure specified by the OMG, a consortium founded in 1989 to promote the adoption of standards for managing distributed objects. Simply stated, the purpose of CORBA is to (1) provide a simple application programming environment that hides the details of using the operating system services, and (2) provide a common application programming environment across multiple computers and operating systems. CORBA is an open standard, and is not bound to an implementation. With CORBA, the OMG's ultimate goal is to achieve object-oriented standardization and interoperability.

CORBA is a *middleware* environment. Middleware is software that lies between application programs and operating system services. In particular, middleware pro-

vides the basic infrastructure for *3-tier* architectures. The key characteristic of 3-tier architectures is the division of a distributed computing environment into presentation, functionality, and data components, such that there is a well-defined interface between each component, and the software used to implement each component can be easily modified or replaced.

1.4.1 The CORBA Object Model

This section presents some background and definitions related to the CORBA object model. The terminology may differ from other classical object models, but the underlying concepts are similar.

- A *client* is an entity — not necessarily an object — that requests a service from an object.
- An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. An object has *state*, *behavior*, and *identity*. The state of an object encompasses all the properties of the object and their current values. Behavior defines how an object acts and reacts in terms of its state changes and message passing. Identity is the property of an object that distinguishes it from all other objects (of the same type). The process of creating an object is called *instantiation*.
- The concept of *interface* is closely related to the concept of class in classical object models. An interface is a description of a set of possible operations that a client may request from an object. An object satisfies an interface if the object can be specified as the target in each potential request described by the interface.
- Interface *inheritance* is a relationship that lets interfaces inherit the structure of their base interface(s). It provides the composition mechanism that permits an object to support multiple interfaces. Interface inheritance does not imply implementation inheritance.
- An *object reference* is a value that denotes a particular object. Specifically, an object reference identifies the same object each time the reference is used in a request. Unlike references of common object-oriented languages such as C++ [Str97], Java [GJS96], or Smalltalk [GR83], a CORBA reference is not limited to a single address space.
- An *operation* denotes a service that an interface provides to its clients.
- A *request* is a message issued by a client to request a service. Conceptually, the object interprets the message to decide what service to perform. In most classical object models, a distinguishing first parameter — the operation name — is required in the message, which identifies the operation to be performed; the interpretation of the message by the object involves selecting a method based on this parameter. The information associated with a request typically consists of a target object identification, an operation, and parameters.

- An *attribute* is a property associated with an interface, which is exported to clients. An attribute is equivalent to a pair of operations for reading and modifying some data of an object.

1.4.2 The Object Management Architecture

The CORBA specification is defined in the context of the *Object Management Architecture* (OMA) [OMG98a], an architectural framework for building interoperable, reusable, portable software components, based on open and standard object-oriented interfaces. The OMA reference model identifies and characterizes the components, interfaces, and protocols that compose the OMA. It forms a conceptual roadmap for assembling the resultant technologies while allowing different design solutions.

CORBA objects are specified in the OMG *Interface Definition Language* (IDL). This language is purely declarative, and provides no implementation details. It is used to specify the boundaries of a component and its contractual interface with potential clients. The CORBA specification defines how IDL constructs map to programming languages, such as C [KR77], C++, Java, or Smalltalk. Figure 1.10 shows the five major parts of the OMA reference model:

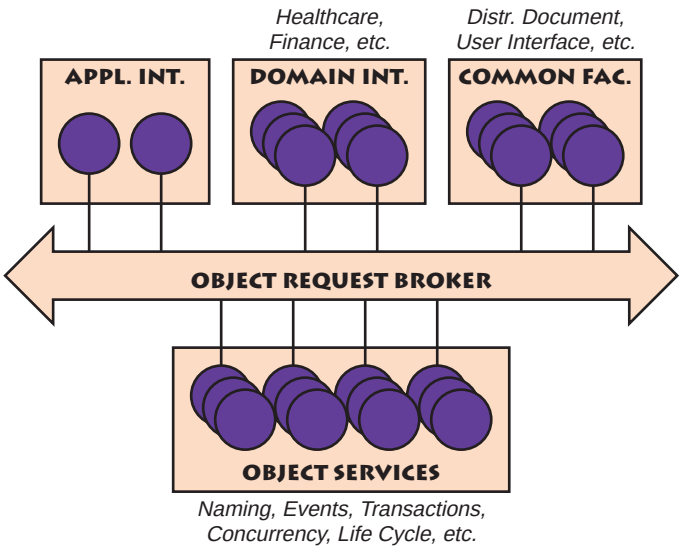

Figure 1.10: The OMA Reference Model

- Commercially known as CORBA, the *Object Request Broker* (ORB) is the communication heart of the standard. It enables objects to transparently and reliably invoke operations on remote objects and receive replies in a distributed environment. The ORB can be viewed as an "object bus", through which heterogeneous objects can interoperate. Integration of distributed objects is available across platforms, regardless of networking transports and operating systems. Compliance with the ORB standard guarantees interoperability of objects over a network of heterogeneous systems.

- The *Object Services* are general purpose components that are fundamental for developing useful CORBA applications. A CORBA service is basically a set of CORBA objects with their corresponding IDL interfaces, that can be invoked through the ORB. Services are not related to any specific application but are basic building blocks, usually provided by CORBA environments, supporting basic functionalities useful for most applications. Several services have been designed and adopted as standards by the OMG [OMG97].

- The *Common Facilities* provide end-user-oriented capabilities useful across many application domains, that can be configured to the specific requirements of a particular application. These are facilities that sit close to the user, such as printing, document management, and electronic mail facilities.

- *Domain Interfaces* represent vertical areas that provide functionality of direct interest to end-users in specific application domains, such as finance or health care.

- *Application Interfaces* are interfaces specific to end-user applications. They represent component-based applications performing specific tasks for a user. An application is typically composed of a large number of objects, some of which are specific to the application, and others part of object services, common facilities, or domain interfaces.

Object-oriented distributed middleware generally *abstracts distribution*, by providing means to invoke remote objects the same way as local objects. It gives the illusion of a continuous address space that contains all the objects involved in a distributed computation. CORBA provides such an abstraction by acting as an ubiquitous *software bus*. It isolates the requesters of services (clients) from the providers of services by a well-defined encapsulating interface. Objects plug on the bus, and export IDL-specified interfaces that act as a contract between clients and servers (this model is usually called *contract-based programming*). Once a CORBA server exports an interface to the bus, clients can ask services from the server. The key concept underlying the CORBA model is the separation of interfaces and implementations, making it possible to abstract object *location* and *implementation* (implementation language, operating system).

Objects are not tied to a client or server role: they can act both as clients and as servers. A program is said to be *CORBA compliant* if it uses only the constructs described in the CORBA specification. An ORB implementation conforms to the CORBA specification if it correctly executes any CORBA compliant program. An application is *portable* if it can be used with different CORBA implementations (by simply recompiling the source code). *Interoperability* is the property of several applications running on different CORBA implementations to interact. The CORBA specification defines a standard protocol for ORB interoperability: the *Internet Inter-ORB Protocol* (IIOP). ORB implementations that use this protocol can interoperate with each other, making it easy to integrate heterogeneous components from different vendors.

The Object Request Broker

The ORB component of CORBA provides more than just messaging mechanisms for remote object invocations. It also provides the environment for managing objects, advertising their presence, and describing their metadata. A CORBA 2.0 ORB consists of several parts, as shown in Figure 1.11:

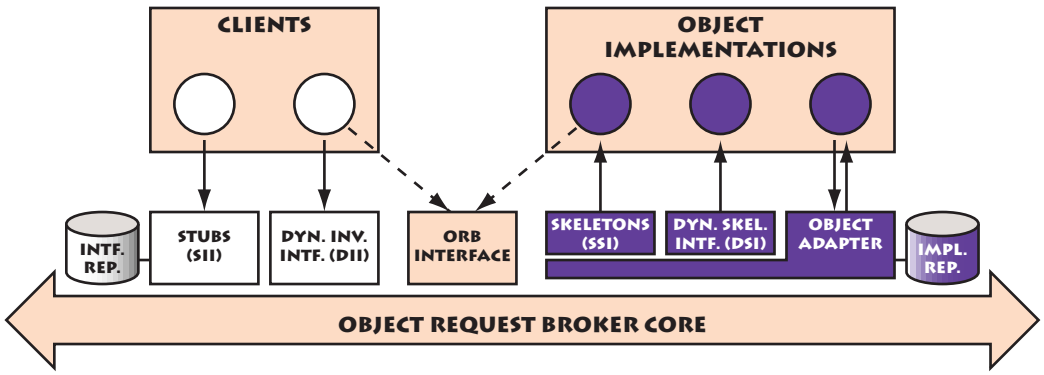

Figure 1.11: The Structure of a CORBA 2.0 Object Request Broker

- The *Interface Repository* (IR) stores interface definitions. It allows the user to obtain and modify the description of component interfaces, the methods that they support, and the parameters that they require. The interface repository allows CORBA components to have self-describing interfaces.

- The *client stubs* — or *Static Invocation Interface* (SII) — provide an interface-specific API for invoking CORBA objects. A client application can invoke a server object through a client stub. From the client's perspective, the stub is a local *proxy* for a remote server object. Stubs are generated by an IDL compiler.

- The *Dynamic Invocation Interface* (DII) allows the creation of requests and invocation of objects at runtime. CORBA defines APIs for looking up the server interfaces, creating requests, generating parameters, issuing the remote call, and getting back the results.

- The *ORB interface* defines a few general APIs to local ORB services.

- The *server skeletons* — or *Static Skeleton Interface* (SSI) — provide static interfaces to server objects. They contain the code necessary to dispatch a request to the appropriate method. Skeletons are generated by an IDL compiler.

- The *Dynamic Skeleton Interface* (DSI) provides a runtime binding mechanism for objects that need to handle requests for interfaces not known at compile time. The DSI is the server equivalent of the DII.

- The *Object Adapter* (OA) provides the mechanisms for instantiating server objects, passing requests to them, and assigning them object references. A standard object adapter called the *Basic Object Adapter* (BOA) must be supported by each ORB implementation. The new CORBA 2.2 specification [OMG98a] introduces a new *Portable Object Adapter* (POA) that overcomes many of the BOA limitations.
- The *Implementation Repository* stores ORB-specific details about object implementations, their activation policy, and their identity.

1.5 Object Groups in CORBA

The idea of adding group communication in an object-oriented middleware is not new. When we started this work (early 1996), at least two products (Electra [Maf95] and Orbix+Isis [II94]) were under development to support group communication in a CORBA-based environment. This section introduces the different design alternatives for managing object groups in CORBA, evaluates them, and describes how they have been used by existing systems.

1.5.1 A Coarse Classification

The CORBA object model defines an object as an entity with a well-defined interface that may be remotely invoked using an *object reference*. An object reference is an *"object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references"* [OMG98a]. This means that the CORBA specification does not permit an object reference to designate a set of objects, and it does not provide ways for clients to invoke several objects at once using an object reference. CORBA only deals with point-to-point remote invocations.

The absence of mechanism that permits the multicast of requests to groups of CORBA objects complicates the design and implementation of many applications that have requirements for reliability and high-availability. Only the CORBA *Object Transaction Service* (OTS) provides fault tolerance to a limited extent through transaction mechanisms, but unlike group-based systems, it does not achieve high availability.

During the last couple of years, several systems have been developed to augment CORBA with groups. We have classified these systems according to three main categories, each of which represents a different approach to group communication in CORBA [FGG97]:

1. The *integration approach* integrates an existing group communication system within an ORB.
2. The *interception approach* intercepts messages issued by an ORB and maps them to a group communication toolkit.

3. The *service approach* provides group communication as a CORBA service beside the ORB, and was chosen as the basis of this thesis.

We now describe these three approaches and existing systems that implement them. We also discuss a variant of the service approach, which consists in adapting the interfaces of an existing CORBA service — the event service — rather than defining new interfaces for group communication.

1.5.2 Integration Approach

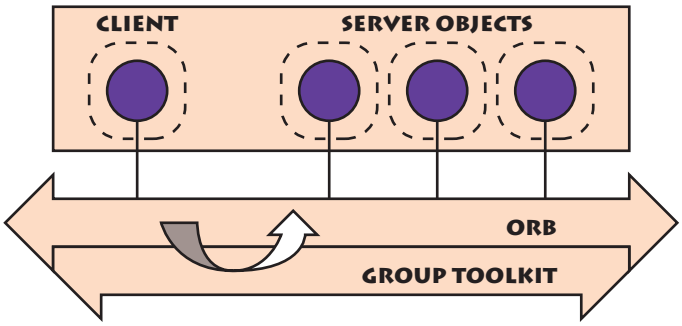

Figure 1.12: Integration Approach

With the integration approach (Figure 1.12), the ORB functionality is enhanced by a group communication toolkit. The ORB directly deals with object groups and references to object groups. CORBA requests are passed to the group communication toolkit that multicasts them from clients to replicated servers, using proprietary mechanisms. The group toolkit is "integrated" into the ORB.[3]

In existing systems that use the integration approach (Orbix+Isis and Electra), only one of the replies is returned by default to the client in order to keep the group invocation transparent.[4] However, a client aware of the group is also able to access all the replies, if necessary. The basic idea is to extend the IDL language mapping and to generate two types of functions from IDL definitions: (1) standard functions that conform to the language mapping and (2) special functions with sequences of values for *out* and *inout* parameters. The client uses the function with the signature that corresponds to its needs. If the client is not aware of groups, it uses only standard functions.

An interesting variant of the integration approach, which has not been explored by existing systems, would consist in providing a second object adapter in addition to the *Basic Object Adapter*. We would then have the basic adapter for standard objects and a dedicated adapter — a *Group Object Adapter* — for group member

[3]Notice that most existing group toolkits are not adapted to a CORBA environment: they are designed for process groups instead of object groups, and they do not provide adequate primitives for group-to-group communication, making it difficult to support client replication [GFGM98].

[4]This makes sense if groups are used for replication.

objects. This approach would be more compliant with the CORBA specification, since it would isolate all group management functionalities in a single non-standard component.

It is interesting to note that Isis Distributed Systems proposed to integrate the notion of object group in the CORBA 2.0 specification [Inc93], but this proposition was rejected by the OMG.

Electra

Electra [Maf95] is a complete CORBA programming environment written in C++. It provides intrinsic support for group communication, supporting the implementation of reliable distributed applications. Electra objects can be replicated to achieve fault tolerance. Electra can be configured for various communication subsystems such as Horus and Isis, and hence can exploit the powerful primitives provided by these subsystems. This means that Electra inherits the advantages and drawbacks of the group toolkit on which it is built. Electra is composed of a set of C++ libraries and daemon applications.

Electra extends the *Basic Object Adapter (BOA)*, adding the ability to *create*, *destroy*, *join* and *leave* a group. Since the modification is performed at the BOA level, every CORBA object of the system can potentially be member of a group. Notification of *view changes* and support for *state transfer* are added to the skeletons in the language mapping. This model is simple and consistent in the sense that all server objects are handled in the same way. However, it adds unnecessary complexity to objects that are not meant to be member of a group.

Electra permits the grouping of object implementations, that support the same IDL interface and run on different machines, into an object group. A CORBA object reference can be bound to an object group and requests are transmitted by reliable multicast. All communication is performed using a toolkit that supports group communication (Isis or Horus).

Electra supports transparent and non-transparent multicast invocations by generating two types of methods from IDL operations. In transparent mode, an object group appears to the client as a highly available singleton object. Non-transparent communication permits programmers to access individual replies from group members by returning all replies resulting from an invocation. Electra clients can specify the group policy to be used for multicast communication with the members of the group.

Orbix+Isis

Orbix+Isis [II94] is a commercial system from Iona Technologies and Isis Distributed Systems.[5] It maps replicated CORBA objects to Isis groups, and provides group communication extensions to Orbix.

[5] Orbix+Isis has been recently discontinued.

With Orbix+Isis, objects that can be group members have specific properties. Unlike Electra, Orbix+Isis sends point-to-point communication through Orbix, and only multicasts are handled by Isis. To handle object replication, the programmer can choose between two execution styles: *active replica* and *event stream*. The role of the server is fixed at compile time by having the server inherit — at the implementation language level — from an abstract base class providing the necessary support for the chosen execution style. Orbix+Isis object groups provide fault tolerance and load balancing through replication.

The active replica execution style offers three communication styles to control how requests are handled:

- The *multicast* style invokes the operation request on all objects of the group.
- The *client's choice* style, useful for read-only methods, invokes the operation request on only one object in the group, based on the client's choice.
- The *coordinator/cohort* style implements primary-backup replication. Only one object of the group processes the request and updates the other objects.

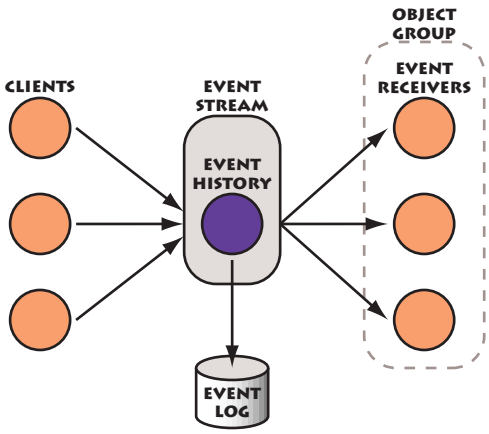

Figure 1.13: Orbix+Isis Event Stream Execution Style

The event stream execution style supports asynchronous requests to object groups using a publish/subscribe paradigm. Clients send messages to servers without receiving replies. Event stream objects have less overhead than active replica groups, and are thus highly scalable. The event stream itself is fault tolerant because it is replicated. The event stream execution style decouples clients from servers and can provide persistence for events, that are kept in the history by the event stream. Servers may register for specific types of events by distinct event streams. The event stream execution style is illustrated in Figure 1.13.

The Orbix+Isis system is made of an application runtime to be linked with clients and servers, and of two types of daemon programs: the Orbix daemon `orbixd` and the Orbix+Isis daemon `isrd`. The Orbix daemon stores information about CORBA

servers (in the implementation repository), and redirects client requests to these servers. The role of the Orbix+Isis daemon is to store configuration information, to authenticate client requests, and to manage group communication. An Orbix+Isis daemon must be launched on each host that runs an Orbix+Isis server.

Orbix+Isis stores information that describes the behavior of object groups in the *Isis repository (IsR)*. The client runtime reads this repository and configures the communication layer based on the server configuration. The configurable elements of a server are *object group details* and *operation details*. Operation details allow the specification of communication characteristics — such as the communication style of active groups and the number of replies to wait for — independently for each operation, while object group details specify global group characteristics.

1.5.3 Interception Approach

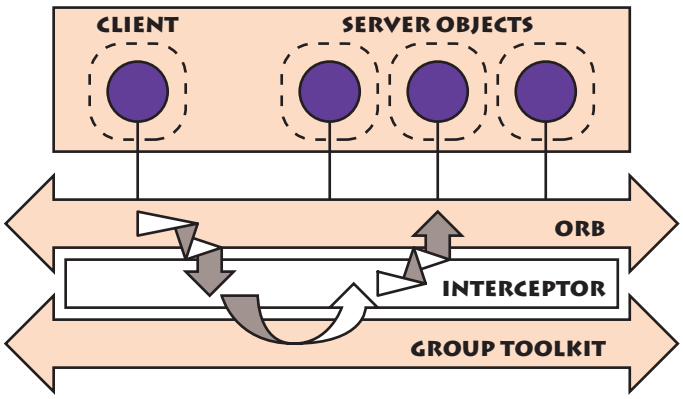

Figure 1.14: Interception Approach

With the interception approach (Figure 1.14), the ORB is not aware of replication. ORB requests formatted according to the IIOP protocol are intercepted transparently on client and server sides using low-level interception mechanisms; they are then passed to a group communication toolkit that forwards them using group multicasts. This approach does not require any modification to the ORB, but relies on OS-specific mechanisms for request interception.

Eternal

Eternal [MMSN98] extends CORBA with capabilities for object replication. It operates on top of the Unix operating system, and works with standard CORBA implementations. Eternal replicates objects, maintains the consistency of replicated objects, and distributes objects across the system. Both client and server objects can be replicated. The degree of replication, the type of replication, and the location of the replicas are hidden by the object group abstraction, with replicas of an object being members of the same object group.

Eternal exploits the Unix `/proc` interface to monitor the operating system calls made by an object to establish an IIOP connection over TCP/IP, and to communicate IIOP messages over that connection. Eternal intercepts the IIOP messages before they reach TCP/IP [NMMS97a], and passes them to the Totem group communication system [MMSA+96], which multicasts the messages to the object groups containing the replicas. Any group communication system that provides guarantees similar to those of Totem can be used instead.

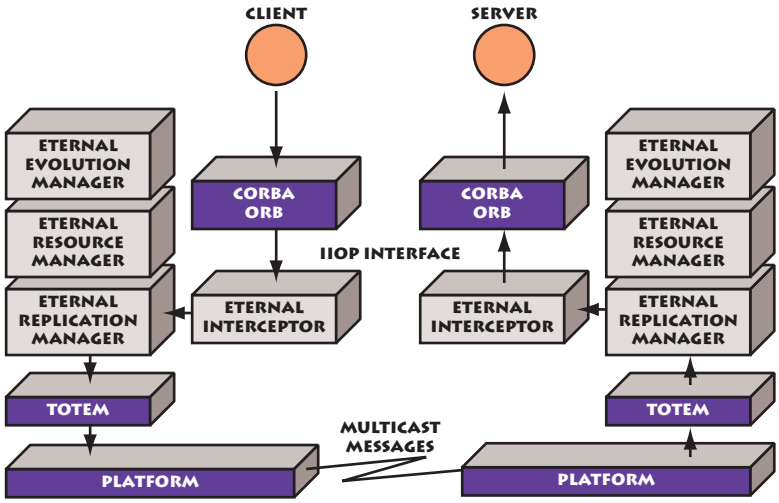

Figure 1.15: The Eternal Architecture

Figure 1.15 shows the different Eternal components. Eternal is composed of an *Interceptor* that intercepts standard CORBA requests, a *Replication Manager* that handles communication with the copies of replicated objects, a *Resource Manager* that manages the system configuration (type and number of replicas), and an *Evolution Manager* that performs the automated upgrade and evolution of objects at runtime. Eternal provides both active and passive replication.

In passive replication, when a client invokes an operation on a server object group, Eternal multicasts the operation to the server object via Totem and only one of the server replicas — the *primary* — actually performs the operation. The message containing the invocation is retained at each of the other replicas, so that they can handle the request if the primary fails. At the end of the operation, Eternal multicasts the updated state of the primary to the other replicas and sends the results to the invoking object via Totem. During the operation, the state of the non-primary replicas may differ from that of the primary; however, the state transfer synchronizes replica states at the end of the operation.

In active replication, when a client invokes an operation on a server object group, Eternal multicasts the operation to the server object group via Totem, and each replica then performs the computation. The underlying totally ordered multicast protocol ensures that all the replicas of an object receive the same messages in the same order, and that they can thus perform the operations in the same order. This ordering of operations ensures that the states of the replicas are consistent at the

end of an operation.

1.5.4 Service Approach

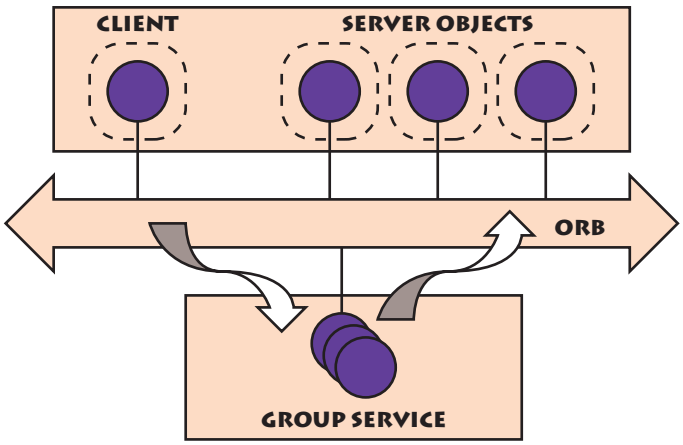

Figure 1.16: Service Approach

The service approach, which has been adopted and developed in the context of this thesis, provides explicit group support through a CORBA service [FGG96] (Figure 1.16). Unlike the integration approach, a CORBA service is mostly specified in terms of IDL interfaces, and does not depend on implementation language constructs. The ORB is not aware of groups, and the service can be used with any compliant CORBA implementation. The service approach complies with the CORBA philosophy, by promoting modularity and reusability. Group support may be provided by adapting an existing CORBA service, or by defining a new service for object groups, as we did in this thesis. Both variants are described in this section.

Notice that a service does not have to be centralized. It can be made of several objects, located at different hosts on the network, that work together at providing the complete service. This is important when dealing with fault tolerance, since a centralized service would be a single point of failure.

Reusing OMG's Event Channels

When adopting the service approach, one might wonder whether an existing CORBA service does already provide a suitable paradigm for object replication, i.e., if the IDL interfaces of an existing service could be reused. We consider here specifically the CORBA event service, which provides a one-to-many communication model potentially reusable for replication.

OMG's Event Service. The CORBA event service decouples the communication between *suppliers* and *consumers* through *event channels*. Suppliers produce

event data and consumers process event data. Suppliers can generate events without knowing the identity of the consumers. Conversely, consumers can receive events without knowing the identity of the suppliers. An event channel is an intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously. An event channel is both a consumer and a supplier of events, and can thus produce or consume events from another event channel.

Event data are communicated between suppliers and consumers by issuing standard CORBA requests. Events themselves are *not* CORBA objects because the CORBA object model does not currently support passing objects by value. There are two approaches to initiating event communication between suppliers and consumers. These two approaches are called the *push model* and the *pull model*.

The push model allows a supplier of events to initiate the transfer of event data to consumers. The pull model allows a consumer of events to request event data from a supplier. In the push model, the supplier is taking the initiative; in the pull model, the consumer is taking the initiative.

Event channels are standard CORBA objects that allow consumers and suppliers to exchange events. Communication with an event channel is accomplished using standard CORBA requests. An example configuration of the event service is illustrated in Figure 1.17.

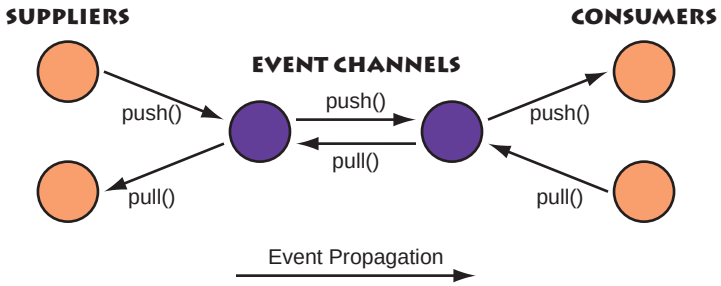

Figure 1.17: Event Service Example Configuration

The basic event service model only allows the sending of data of type `any`.[6] Hence, it provides a message-passing-like interface to event communication. The event service also provides a *typed* model that offers RPC-like communication. Suppliers call operations on consumers (typed push model) or consumers call operations on suppliers (typed pull model) using operations of an application-specific IDL interface.

Replication with Event Channels. The event service provides a natural way to multicast requests to replicated objects, using the *push* model with *one* event channel: all the copies of a replicated object are consumers of the channel, while clients supply event data on this channel (Figure 1.18).

In particular, the typed version of the event service provides for straightforward transparent server replication: clients register with the event channel as suppliers

[6] In CORBA, `any` variables can contain data of any type.

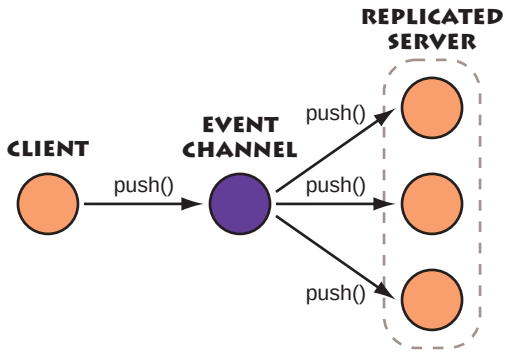

Figure 1.18: Using the Event Service for Replication

and obtain a reference to an object that supports the same interface as the replicated server; they can then issue invocations on this interface directly. Clients do not need to know the number or location of the copies, which can change at runtime.

Limitations of the approach. Using event channels for replication has some major limitations. There are no group management facilities, nor guarantees concerning ordering, atomicity, or failures [DFGG97]. For instance, strict atomic delivery between all suppliers and all consumers requires additional interfaces [OMG97]. Different qualities of service may be provided by different implementations, but they are not standardized in the event service.

In addition, the model of the event service is not ideal for object replication. The event service only supports one-way communication, i.e., operations on the replicated server must have only *in* parameters. This restriction is too limited for many distributed applications that require results from invocations. Return values may be transmitted using a "reverse" event channel, but this requires clients and servers to be both consumers and suppliers.

Another fundamental problem is that the design of the event service is *centralized*. Although consumers and suppliers use different interfaces for pushing and pulling event data to and from the channel, they have to invoke the same centralized object in order to connect to an event channel. This event channel is a standard CORBA object, which is a single point of failure in the event service architecture. Since our aim is to provide fault tolerance through replication, we cannot rely on a solution that introduces a single point of failure. There are several ways for decentralizing the event service [FGS97]. The solution that we consider the most promising consists in chaining event channels, and is shortly described below.

Rather than having a single event channel that diffuses messages to all copies of the replicated object, we introduce several event channels located on the client and on the server site. A client is represented as a *request-supplier* and a *response-consumer*, while a server is represented by a *request-consumer* and a *response-supplier*[7] (Fig-

[7] These event channels reside typically in the same process as the client and the servers.

ure 1.19). This model provides two-way communication with no single point of failure. Distinct clients generate data using distinct request-suppliers and receive replies through distinct response-consumers.

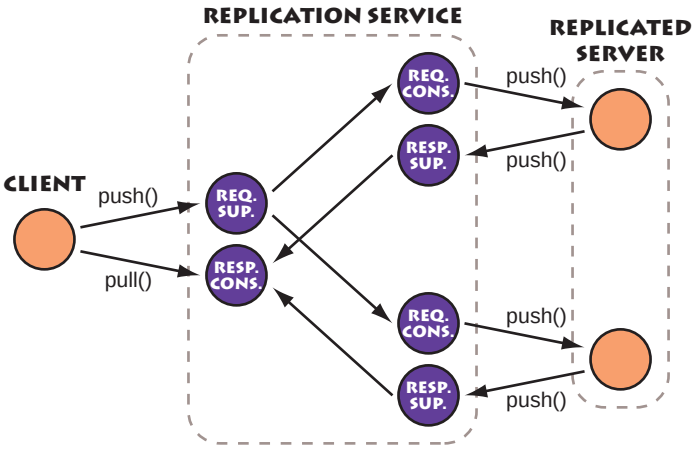

Figure 1.19: Decentralizing Event Channels using Chaining

The request-supplier object performs multicast communication, possibly executing some protocol with the request-consumers for guaranteeing atomicity and ordering of messages. The response-consumer gathers multiple replies and returns them to the client through a push or a pull mechanism.

This approach is fully CORBA compliant, does not modify the event service specification, and does not depend on a single point of failure. Extra protocols are however necessary between the channel objects for ensuring atomicity and ordering of messages, and also for group membership.

Specifying a New CORBA Service

We have elected to specify a new service for object group support in CORBA [FGG96]. This service, named the *Object Group Service* (OGS), is the core of this thesis. We have chosen this approach because none of the existing CORBA services offers the right abstraction for group communication. Dealing with object groups requires specialized interfaces for group management and group communication, with special mechanisms to ensure quality of service.

CORBA's open architecture allows us to easily define and implement new services. The process of specifying a new service consists in isolating the requirements, choosing the right abstractions, and specifying the interfaces for these abstractions. The OMG has published guidelines for designing object services and their interfaces [OMG97]. These guidelines are summarized in Section 2.2.1, and have been followed during the design on OGS. As stated in Chapter 5.6, the OMG has recently issued a *Request For Proposals* (RFP) titled: *Fault Tolerant CORBA Using Entity Redundancy* [OMG98b]. This RFP invites proposals for new CORBA functionalities for replicating entities using group mechanisms, just as we did in OGS.

1.5.5 Evaluation of the Different Approaches

This section presents an informal comparison of the three different approaches. This comparison is based on implementations of these approaches (Electra, Orbix+Isis, Eternal, the CORBA event service, and OGS). It should not be considered as an exhaustive survey of the pros and cons of each approach.

Approaches Comparison	**Integration**	**Interception**	**Service**
Transparency	+	+	
Ease of Use	+	+	+
Portability	–		+
Interoperability		+	+
Modularity	–		+
CORBA Compliance	–	+	+
Performance	+	+	
Simplicity	–		+

Table 1.1: Comparison of the Different Approaches

We focus on several different aspects: transparency, ease of use, portability, interoperability, modularity, CORBA compliance, performance, and simplicity. Table 1.1 summarizes how the integration, interception, and service approaches fit these criteria. In the table, + (plus) means good, – (minus) means limited, and blank means satisfactory. Notice that some of our judgments may have been influenced by *implementations* of the approaches, rather than by the approaches themselves. The following explains the criteria and justifies our judgments.

Transparency

Transparency hides groups to the programmer, by giving the illusion that the invocations are issued to and originated from single objects.

The integration approaches provide *client transparency*.[8] The invoker does not need to know that the invokee is a group, although it can benefit from this knowledge.

The interception approach (as provided by Eternal) *enforces* transparency. In contrast with the other approaches, it does not allow, for instance, a client to access all replies of a multicast invocation.

The service approach can be configured with or without transparency. Transparency is provided by the *typed* event service and the *typed* object group service, as described later in this dissertation.

[8] Full *server transparency* does not make much sense with dynamic groups, because member objects must provide explicit support for transferring their (application-specific) state. However, in some situations, state transfer may be replaced by keeping copies of all the messages sent to the group since its creation, and forwarding these messages to the new members.

Ease of Use

Ease of use is an important consideration since it shortens program development time and it can make the application more robust and reliable, by reducing chances for programmers to err.

Transparent group support (integration and interception approaches) is easier to use, since it requires no explicit construct on the client side. The client does not care whether the server is a group or not. On the other hand, using advanced features of group communication (e.g., associating particular semantics with a multicast invocation) is generally more complex with transparent approaches since it is not orthogonal to invoking groups.

The event service provides a familiar publish/subscribe programming model; its interfaces are already standardized and well-known, making them easy to use by many programmers. Our object group service combines the advantages of explicit group management and transparent group invocations in a CORBA service; application can thus choose between powerful configuration facilities and easy-to-use transparent group support.

Portability

With portability, we measure how independent a software component is from a specific ORB or architecture. We focus on two types of portability: *group support code* (i.e., the code responsible for group management and group communication) and *application code* portability.

Electra and Orbix+Isis are built around their own ORBs, and integrate group support code within the ORB. Therefore, this code is clearly not portable to other architectures. In addition, the integration approach adopted by Electra and Orbix+Isis uses non-standard language-specific constructs,[9] causing application code to be also not portable.

Eternal makes use of Unix low-level features and its group support code is not portable to all architecture. Application code is independent of Eternal, and is thus fully portable.

With the service approach, both the group support and application code are portable to any CORBA compliant architecture, assuming that the code does not depend on implementation-specific ORB feature. Unlike the approaches that make use of group communication toolkits, it is not limited to the architectures supported by the toolkit.

[9] These constructs are quite different in Electra and Orbix+Isis; application code written for one is not portable to the other.

Interoperability

Implementations based on group toolkits that make use of proprietary communication protocols are not interoperable. This is the case of Electra. Orbix+Isis combines invocations through Isis and through IIOP, and can thus interoperate *with singleton objects* through point-to-point IIOP invocations. Eternal may choose which request it intercepts, and can interoperate through IIOP as well.

The service approach uses only the communication primitives of the underlying ORB, and is thus fully interoperable.

Modularity

Modularity enables the reduction of the complexity of a system by partitioning it into a set of cohesive and loosely coupled components. Each of these components may be modified, extended, or replaced without requiring any change to other components, thus making the system easier to maintain. The modularity of an approach is difficult to evaluate, since it involves implementation considerations. But a general rule is that a system with decoupled components is more modular than a monolithic system.

The integration approach provides a monolithic architecture, in which group support is part of the ORB. Therefore, modifying part of the system (e.g., using another group communication toolkit) may require updating the whole system.

The interception approach decouples replication from the ORB itself, thus preserving the system's modularity. But the replication infrastructure of Eternal is not packaged as a set of components independently reusable by CORBA applications.

The service approach promotes reuse and modularity by defining a new component with IDL-specified interfaces, that can be integrated with other services or applications. Furthermore, our object group service is composed of several independent IDL-specified CORBA services, none of which is exclusive to group communication.

CORBA Compliance

A client or server program is said to be *CORBA compliant* if it conforms to the CORBA specification. The integration approach is not fully compliant since it modifies and extends the CORBA specification. The ORB core has to deal with references to replicated objects, and the semantics of CORBA references are modified. Furthermore, both Electra and Orbix+Isis use language-specific constructs and extend the C++ mapping specification.

Both the interception and service approach are compliant (assuming that they do not rely on implementation-specific ORB features). The interception approach is completely decoupled from the ORB and only relies on IIOP constructs. The service is independent of the ORB core as it is used only through IDL-defined interfaces, and does not make assumptions about the underlying ORB implementation.

Performance

Systems that provide high-level abstractions (such as flexibility, modularity, and interoperability) are generally less efficient than low-level systems. In the context of group communication, performance also depends directly on the protocols used for multicast invocations, as well as the quality of service provided by these invocations.

The integration approach is the most efficient, since communication can be optimized in the ORB itself. There is no indirection when invoking servers: only one type of communication (group multicasts) is used.

The efficiency of the interception approach depends both on the ORB (time required for IIOP communication) and the underlying group communication toolkit. In this case, two types of communication are used: standard IIOP requests and group multicasts. Therefore, two indirections are required when a client invokes a group (IIOP is mapped into a group multicast on the client side, and transformed into IIOP on the server side).[10]

The service approach uses the communication primitives of the ORB on which it runs. Therefore, its efficiency depends directly on the underlying CORBA implementation. Two indirections are also required when a client invokes a group.

Simplicity, Weight, and Adequacy

We consider an approach to be simple and lightweight if it is adequate for the problem and performs what it is meant to do, without overloading the system with unnecessary features. The OMG guidelines state that a service should be designed to do one thing well and to be only as complicated as it needs to be.

The current implementations of the integration and interception approaches make use of external group communication toolkits for multicast invocations; most of these toolkits do not provide adequate support for *object* groups [GFGM98] (they deal with *process* groups), and software layers must be added for interfacing them with the CORBA world. In addition, very few of these toolkits allow group-to-group communication; this communication model is necessary for grouping CORBA objects that act both as client and server.

In contrast, the service approach, as defined by our object group service, is lightweight and adequate for the problem; it provides only the required primitives for group communication, and is built as an independent, optional CORBA component. Furthermore, it may be configured to share resources between several application processes (see Section 5.1.1).

[10] Note that IIOP requests are intercepted *before* they reach the network, which makes the cost of this extra indirection generally negligible.

Summary

A distributed computing system is a set of tightly coupled programs executing on several hosts, that cooperate to perform a task that none could carry out in isolation. Such systems have become commonplace since the popularization of the Internet. However, managing distributed systems is a difficult task since one must deal with remote communication and the various types of failures that may result from distribution.

Several object-based distributed frameworks have appeared to simplify the development of distributed applications. These frameworks provide high-level abstractions that encapsulate distribution and hide remote communication from the application developer. However, they only deal with invocations to individual objects, while several kinds of applications require one-to-many communication. An interesting extension to these frameworks consists in augmenting them with group communication primitives. The key idea of group communication is to gather a set of processes or objects into a logical group, and to communicate with all group members at the same time with various ordering guarantees. Several group communication toolkits are available, but most of them are inadequate for deployment in object-based frameworks.

Adding group communication to a middleware environment like CORBA allows the application developer to benefit from the power of groups (high availability, fault tolerance, etc.) while preserving the key features of the middleware environment (simple development process, distribution transparency, component integration, etc.). There are several different design alternatives for managing object groups in CORBA: the *integration approach* integrates an existing group communication system within an ORB; the *interception approach* intercepts messages issued by an existing ORB and maps them on a group communication toolkit; and the *service approach* provides group communication as a CORBA service beside the ORB. The service approach best matches the CORBA environment since it encapsulates group support as an independent, well-defined component.

Chapter 2

The Object Group Service: Concepts and Overview

The technique of mastering complexity has been known since ancient times: "divide et impera" (divide and rule).

E. Dijkstra

This chapter gives an overview the CORBA *Object Group Service* (OGS) developed in the context of this thesis. We describe the OGS approach based on component integration, and explain how these components comply with the CORBA design guidelines. We present the OGS group model, and how it encapsulates plurality and behavior. We define the concept of transparency in the context of object groups, and we present the mechanisms used to provide it. We finally give an architectural overview of the OGS components, which are detailed in the next chapter.

2.1 What is OGS?

The *Object Group Service* (OGS) is a CORBA service that provides group management and communication facilities in a CORBA environment. It is composed of a set of generic IDL-specified interfaces. With OGS, clients can send invocations to object groups without knowing the number and identities of group members. In addition, OGS provides support for *transparent group invocations*, allowing clients to invoke operations on object groups as if they were invoking singleton objects. OGS is based only on standard CORBA mechanisms and is thus portable to any compliant ORB implementation. OGS may be used from any programming language that is supported by CORBA, or from any system that supports the CORBA *Internet Inter-Orb Protocol* (IIOP).

More precisely, the OGS environment specifies an architecture and a set of IDL

interfaces for object group support, as well as a set of object services that provide various facilities for reliable distributed computing. These services, which include distributed agreement protocols and detection of remote component failure, are used in the implementation of the group communication primitives. The relations between these different services are *implementation dependencies*, and are not expressed in the IDL interfaces. The implementation of OGS, described in this dissertation, should be seen as a proof of concept, and not as an authoritative reference.

In this dissertation, the term "OGS" designates both the single CORBA service that defines the interfaces for group communication, and the environment composed of all the individual services specified in this thesis and used for the implementation of the group communication primitives. When the context is ambiguous, we will use the term "OGS environment" or "OGS framework" to explicitly denote the latter.

2.1.1 A Component-Oriented Approach

A CORBA *component* is a building block that employs object-oriented mechanisms like inheritance and polymorphism to provide a specific service. A component is a unit of work and distribution, and it generally designates an object or a set of collaborating objects that may be accessed by clients through a well-defined interface.

The CORBA distributed object infrastructure makes it easier for components to be autonomous, self-managing, and collaborative. CORBA's distributed object technology allows us to put together complex client/server information systems by simply *assembling* and *extending* components: a client/server application becomes a collection of collaborating components. Individual objects may be modified without affecting the structure of the other objects in the system, or the way they interact. The standardized object bus is used for *heterogeneous component integration.*

OGS uses a component-oriented approach. Basic units of functionality are packaged as CORBA services, which represent distributed components. These services are specified independently from each other, and may be used in isolation. This modular decomposition makes OGS an *object-oriented framework of CORBA components for reliable distributed systems.*

2.1.2 The OGS Approach vs. Protocol Frameworks

In contrast with protocol frameworks like Bast [Gar98], OGS services are black-box style components and do not directly address protocol design. The CORBA component-oriented approach followed by OGS separates completely the interface description from the actual implementation. This approach allows the programmer to develop his components with different programming languages, and yet have these components cooperate together. However it makes it difficult to reuse *implementations* through inheritance mechanisms as in white-box frameworks. Such frameworks do not match the CORBA model, since they are not appropriate for language heterogeneity.

Nevertheless, the architecture of OGS is modular and flexible enough to allow ex-

tensions and reuse of individual components. A protocol developer may modify a component *in isolation*, and replace it in the OGS architecture without impact on the other components. Furthermore, since the architecture of OGS is specified only in terms of IDL interfaces, implementations are free to use frameworks of objects and patterns [GHJV95] similar to Bast internally.

2.2 From Distributed Objects to CORBA Services

Whereas a distributed object is a single identifiable entity located somewhere in the system, a service may be composed of any number of objects, located on different machines, and that work together at providing a complete service: it is a building block that provides an effective solution to a class of problems using a set of collaborating objects. A CORBA service is defined by one or several IDL interfaces and the specification of their behavior.

2.2.1 CORBA Service Guidelines

The CORBA specification describes several principles for designing object services and their interfaces [OMG97]. This section summarizes the principles that influenced the architecture and design of OGS. We will come back to these points in Chapter 3.

- *Build on CORBA concepts:* services must be designed based on the concepts introduced by the CORBA model, such as separation of interface and implementation; object references are typed by interfaces; clients depend on interfaces, not implementations; use of multiple inheritance of interfaces; use of subtyping to extend, evolve, and specialize functionality. These concepts allow us to build powerful services that comply with the CORBA design guidelines.
- *Basic, flexible services:* services should be designed to do one thing well, and be only as complicated as they need to be. These simple services can then be combined together to provide powerful functionality.
- *Generic services:* services must be designed so they do not depend on the type of the client object nor, in general, on the type of data passed in requests. They should be as generic as possible in their interfaces, and should preserve *maximum implementation flexibility.*
- *Allow local and remote implementations:* services should be structured as CORBA objects that can be accessed locally or remotely. This allows considerable flexibility as regards the location of participating objects. In addition, it provides support for very efficient implementations based on a library object adapter that enables execution of service objects in the same process as the client.
- *Objects often conspire in a service:* a service is typically composed of several distinct interfaces that provide different views for different kinds of clients of

the service. Implementations can support the service interfaces as a single CORBA object or as a collection of distinct objects, allowing considerable implementation flexibility.

- *Use of callback:* callback interfaces are interfaces that a client object is required to support to enable a service to call it back to invoke some operation. The interfaces clearly define how a client object participates in a service, using standard IDL and operation invocation mechanisms.

- *Use of exceptions and return codes:* exceptions should be used only for handling exceptional conditions such as errors. Normal return codes should be passed back via output parameters.

- *Use of interface inheritance:* services should use interface inheritance (subtyping) whenever the client code requires less functionality than the full interface. Services are often partitioned into several unrelated interfaces if it is possible to partition the clients into different roles.

- *Use of object factories for remote object creation:* creating an object in a remote address space is not supported by language constructs. In CORBA, this is performed using object factories, which are CORBA objects located in the remote address space that create object instances on behalf of clients.

2.3 The OGS Model

While objects have a well-defined meaning in the context of object-oriented systems, the semantics of object groups in distributed systems is more diffuse and varies depending on the context. The concepts of encapsulation and invocation, for instance, do not have the same meaning when dealing with object groups instead of individual objects. This section introduces the object group model adopted for OGS, defines the major concepts related to object groups, and presents how groups may be used to encapsulate plurality and behavior.

2.3.1 Object Groups

An *object group* is a set of objects logically related. They may or may not act as copies of a replicated object. No assumption is made about the location and number of these objects. A group acts as a *logical addressable entity*. We call the objects that are member of a group *member objects* or simply *members*. An entity that requests a service from a group is a *client* of the group.

In the context of replication, a *replicated object* O is a set of individual objects O_i, which act as copies of the same object O. These individual objects are called *replicas*.

2.3.2 Group Behavior

We define *group behavior* as the set of actions taken by the members of a group, e.g., for diffusing and handling client requests. It encompasses the way member objects are coupled and how they cooperate. In this respect, active replication with total order invocations, or primary-backup replication are examples of such behaviors.

2.3.3 Group Designation and Group References

Designating an object group requires some addressing facility, i.e., a way of designating the whole group as a single entity instead of a set of individual objects. To cope with this problem, we introduce the notion of *group reference*. The same way as a CORBA *object reference* designates a (possibly remote) object, we define a *group reference* as a value that designates a set of objects; the number and location of these objects is hidden from the entity that holds the reference and may vary over time.

Since group composition may vary over time, there is no built-in notion of group composition in group references. Two group references may designate different groups, although these groups contain exactly the same set of objects, i.e., the references to the individual objects that compose the groups are identical as specified by the CORBA identity model. Two group references are identical only if they correspond to the same group (a group is identified by a unique group identifier).

The definition of group references does not imply any implementation policy. A group reference may be an object (created implicitly or explicitly), a simple type, a string, a language construct, etc. The way group references are implemented depends of factors such as the host environment, or the degree of transparency, flexibility, and efficiency required. In OGS, a group is defined by a unique group identifier — the name of the group — and a group reference is implemented as a CORBA object.

2.3.4 Groups and Encapsulation

A desirable property of an object group is to behave like a singleton object, i.e., to act as an identifiable, encapsulated entity that may be invoked by a client. The key design issue to provide this abstraction consists in hiding the major differences between object groups and singleton objects. OGS does this by encapsulating *plurality* and *behavior*.

Encapsulating Plurality

A group consists of a set of objects that may all be invoked together. In an object-oriented environment, a client holding a reference to an object group should be able to request a service from the group as if it were invoking a singleton object. This means that an object group must be considered as a first class object, which

abstracts the *number* of the objects that actually implement a service. This problem is called *plurality encapsulation* [BI93].

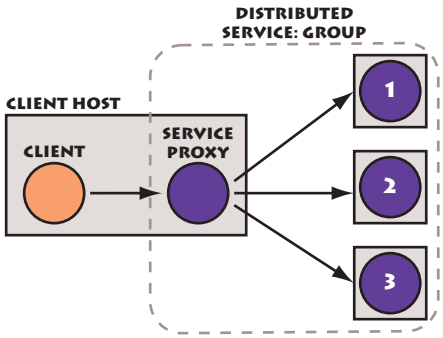

Figure 2.1: A Group Proxy

A general solution to encapsulate plurality is to use the notion of proxy [Sha86]. A proxy is an entity that acts as a local representative for a remote service. To a client, a proxy behaves as if it were implementing the service itself, while it actually forwards the requests to the actual servers and returns the replies back to the client. Figure 2.1 illustrates the concept of proxy with a client invoking a distributed service, composed of three remote objects, through a local proxy.

In OGS, a group reference is a facility for naming and invoking object groups, which encapsulates plurality. In this respect, a proxy is only one example of group reference. Group communication mechanisms of OGS are based on group proxies.

Encapsulating Behavior

Communicating with object groups is very different from communicating with individual objects. Request atomicity and ordering, multiple replies, and partial failures are examples of problems that occur when communicating with object groups. Depending on the application, different mechanisms may be used to solve these problems. These mechanisms define the group's *behavior*. For instance, the replication policy of a replicated server is part of the group's behavior.

From the client's perspective, plurality encapsulation makes an object group *appear* as a singleton object. Similarly, behavior encapsulation makes an object group *behave* like a singleton object. In particular, the way invocations are diffused in the group and the way they are handled is encapsulated by the group. Member objects do not have to behave individually the same way as if they were not replicated (e.g., they can share the work, or have the request handled by only one server), as long as the client receives a consistent reply.

OGS encapsulates the group's behavior in the group itself, by providing transparent mechanisms that allow group members to associate specific semantics to group invocations without the knowledge of clients.

2.4 Transparency in OGS

A common goal of object-oriented middleware environments is to hide the low-level mechanisms used for remote invocations and object management as much as possible, to let the developer focus on the application-specific problems. Similarly, when working with object groups, a desirable property is to hide the complex mechanisms used for group communication. This section describes the concept of transparency in the context of OGS and the underlying mechanisms used to provide it.

2.4.1 The Meaning of Transparency

Transparency is the property of a system to be invisible, i.e., the degree to which application programs are unaware of the system. In this respect, encapsulating plurality and behavior contributes to transparency. Transparency may have different forms. We distinguish the following types of transparency in OGS, classified according to the role that objects have in the system, and their interactions with object groups:

- *Client-side group transparency:* to its clients, a group is not distinguishable from a singleton object that implements the same interface. A client sends a single request and receives a single reply, as it would with a single object. A requirement is that all the objects of the group have the same interface or a common base interface, which is used by the client. Client transparency has the property that any singleton object can be replaced by an object group without change to its clients.
- *Server-side group transparency:* for a server, a client group is not distinguishable from a single object invoking the server. This means that the group has to arrange for only one request to be issued to the server, even though the client group may contain several objects.
- *Member's group transparency:* the member objects need not be aware that they are member of a group. They receive invocations the same way as if they were singleton objects. This transparency type requires some assumptions on properties like determinism of object implementations, and thus is not always possible or desirable to achieve.

Figure 2.2 illustrates the different types of transparency on a replicated account object that has a `deposit()` operation for depositing money on the account. Client-side group transparency is provided by having the clients directly invoke the `deposit()` operation on the account's interface (1). Server-side group transparency is achieved by OGS, which filters duplicate requests from client groups, issuing only one request to each account object (2). Finally, member object's group transparency is provided by having OGS directly invoke the `deposit()` operation of account objects (3).

In addition to belonging to one of these categories, transparency appears at different levels in the OGS architecture [GFGM98] and may also be classified as follows:

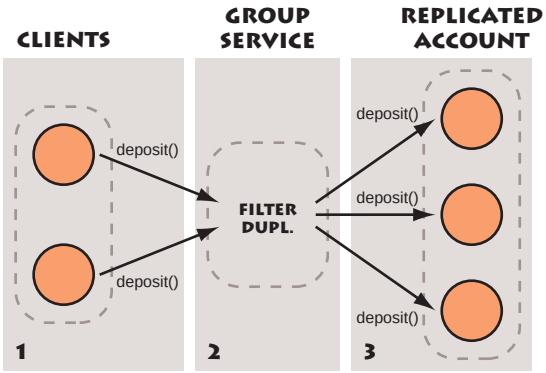

Figure 2.2: Different Types of Transparency

- *Plurality transparency:* the application objects have the illusion that they deal with singleton objects, although they interact with object groups.
- *Behavior transparency:* the application objects are not aware of the replication policy and the protocols run by OGS in order to make the system behave consistently.
- *Type transparency:* requests are performed via direct method invocations on the server's interface, rather than by explicit calls to a group communication API (e.g., via a `multicast()` operation), and explicit packing and unpacking of operation parameters into and from messages. Type transparency means that the application does not need to perform type conversions.

Full transparency is achieved if the application program is completely unaware of OGS. In group communication, transparency is generally limited to replication, since the semantics associated with replicated invocations are clear and simple to hide from the application. But, as we will see in Section 5.4, client-side transparency may be achieved with OGS even when group communication is used for another purpose than replication, such as parallel processing.

2.4.2 The Benefits of Transparency

Transparency provides a number of facilities which make it useful, if not necessary, to develop distributed applications with object groups. Some of the benefits of transparency are outlined below.

- *Ease of use.* Applications do not need to be written with replication in mind.
- *Less error prone.* Application programmers need not write error prone code for inserting and extracting data into and from requests.
- *Reuse of existing code.* Existing client code can be reused without modification. Server code requires only minor modifications. Objects from existing

frameworks can be made groupable by using multiple inheritance of interfaces and implementations.

- *Encapsulate behavior in the group.* OGS knows about the operations invoked on object groups, and can associate different semantics to distinct requests on behalf of the object group.[1] The protocols may vary depending on the invoked operation without the client's knowledge.

2.4.3 The Limitations of Transparency

Client transparency is generally considered as a "good thing" for object replication. However, server transparency has several drawbacks for complex systems. When dealing with replica configuration, failure detection mechanisms, or advanced synchronization between replicas, explicit group management support is required. In some cases, transparent replication can make systems suffer reduced performance due to the lack of control over how replication is performed. Transparent replication is generally considered *less efficient* and *less flexible* than explicit replication.

2.4.4 Support for Transparency

Table 2.1 summarizes the different types and perspectives of transparency, sorted according to the classifications described in Section 2.4. The mechanisms used to provide this transparency are pointed out in the table and described in the rest of this section. Most of these mechanisms are used by OGS to support transparent group invocations.

Transparency	**Client-side**	**Server-side**	**Member**
Plurality	Request diffusion Response collation	Request filtering	
Type	Typed invocation		Typed delivery
Behavior	Server-specified semantics		Consistent delivery

Table 2.1: Classification of Transparency Types

Request Diffusion

When a standard invocation originated by a client and sent to a singleton object actually targets an object group, OGS must transparently diffuse this request to all members of the group, with some level of reliability and ordering. We call this action *request diffusion.*

Request diffusion protocols ensure that all correct member objects actually receive all requests in a consistent order, thus preserving the correctness of the system. In OGS, diffusion is performed to a group of objects whose composition — number and identity of members — may vary during the system's lifetime.

[1] In OGS, this is only possible with transparent invocations.

Response Collation

When invoking a singleton object, a client receives a single reply. For preserving client transparency, OGS must also return a *single* reply to the client as the result of a group invocation, although several replies might be sent by the group members.

We call *collation* the process of collecting a set of messages and producing a single message. When dealing with active replication and fully deterministic servers, collation is trivial since all replies are identical, and OGS can return the first one it receives to the client.

When groups of identical objects are used for parallel processing, with each group member handling one part of the request, the correct semantics of collation is to construct a new reply that consists of the union of all replies. Performing this task in a generic way is a challenging task since it requires knowledge of the reply types and semantics. In particular, only part of the reply data may require aggregation. This problem can be solved with OGS by paying special attention to component interactions when designing the application, as detailed in Section 5.4.

Request Filtering

Similarly to response collation, *request filtering* must be performed when a group of replicated objects invokes another group of objects (or a singleton object), so that target objects do not receive duplicate invocations.[2]

Request filtering can be performed upon request emission (in the invoker group) or upon request reception (by the invokee) [Maz96]. Performing request filtering upon request reception limits filtering to invokees that have knowledge about groups and filtering, and will not work when a client group issues a request to a standard *singleton* object.

Typed Invocation and Delivery

Type transparency is the ability of the group service to know and use type information from the server's interface. It enables a client to directly invoke operations on the server's interface, using static invocation mechanisms (*typed invocation*). The server benefits from type transparency by having the operations of its interface directly invoked by the group service (*typed delivery*).

To provide type transparency, OGS must get some knowledge about the invokee object's interface. While this is trivial in dynamically typed languages such as Smalltalk, doing this in statically typed languages requires runtime knowledge about the server's interface. CORBA provides this runtime support through the *Interface Repository* (IR), and makes it possible to receive and construct operation invocations

[2] Note that request filtering is not necessary if the invoked operation is idempotent, i.e., if invoking the operation more that once does not change the state of the invoked object nor the content of the reply. In this case, multiple invocations are harmless and do not change the behavior of the system, although they consume unnecessary resources.

for an IDL interface not known at compile time through the *Dynamic Skeleton Interface* (DSI) and the *Dynamic Invocation Interface* (DII). This approach is detailed in Section 4.4.1.

Server-Specified Semantics

Whereas traditional group communication toolkits force the *client* to choose the semantics of multicasts (e.g., reliable, total order, etc.), OGS lets the *server* specify the semantics required for each operation. The client does not need to bother with either the semantics of the operation or the protocol required in order to keep the system in a consistent state. This is only possible in combination with typed communication, since untyped communication does not map messages to operation invocations (the operation name is used as a key to decide upon the protocol to execute).

This approach enforces behavior encapsulation as the client does not need to know the ordering guarantees that actually depend on the operation implementation. Furthermore, this approach lets the server optimize concurrent client requests according to their semantics (see Section 3.1.2).

Consistent Delivery

When invocations are issued to group of objects instead of individual objects, OGS has to run some protocols in order to make the system behave consistently. Consistency has different meanings depending on what groups are used for. For instance, with active replication, all group member objects have to receive and process requests from different clients in the same total order to maintain the global consistency of the group of replicas. This ordering requirement may be relaxed if operations do not modify the shared state of the replicated object, or if two operations modify disjoint parts of this state.

Behavior transparency and consistent delivery abstract these considerations from the user application. The application developer can make the assumption that invocations are delivered to the server in a consistent order. It is then possible to modify the underlying protocols, provided that they keep the same semantics, without the application having to know about it.

2.5 OGS Architectural Overview

When developing group communication systems, one has to solve various problems, such as distributed agreement and failure detection. These problems are also addressed in this work. The original aspect of our approach is that they appear as independent CORBA components in the OGS architecture. In fact, defining independent services for these generic problems promotes modularity and reusability, and allows the development of other types of reliable distributing systems, by sim-

ply using the relevant components of the OGS architecture. This section briefly describes the OGS architecture, and how it relates to traditional group communication systems.

2.5.1 Architecture of Group Communication Systems

A group communication system is typically composed of the following components, generally organized in a layered architecture (Figure 2.3):

Figure 2.3: Components of a Group Communication System

1. *Reliable communication* is used to implement higher level protocols. Many systems are based on standard connection-oriented protocols, such as TCP/IP.
2. *Failure detection* is necessary to detect the failure of individual group members, and is typically used for agreement protocols.
3. *Agreement protocols* are protocols used by higher layers for agreeing on common values, such as group composition and message ordering.
4. *Group membership* keeps track of the composition of groups, and allows members to dynamically join or leave groups. It is also responsible for detecting group member failures, updating the view accordingly, and delivering view changes to the application.
5. *Group multicast* provides reliable delivery of messages to all group members, with various ordering guarantees.

2.5.2 OGS Components

OGS maps the layers of the group communication architecture presented above to CORBA services, using a *component-oriented* approach. The services are not organized in a layered architecture, but as a set of orthogonal components with *usage relationships* between each other. Figure 2.4 presents an abstract view of the major components defined in the OGS architecture. Although this is not clearly visible on the figure, each component is specified independently and they all interact with each other through the ORB. The application may use any of these components directly. These components are:

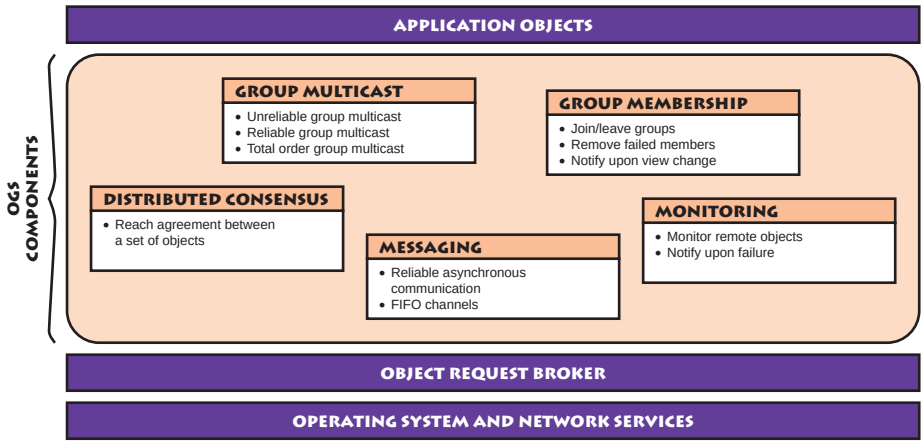

Figure 2.4: Overview of the OGS Architecture

1. A *Messaging Service* that provides non-blocking reliable point-to-point and multicast communication.
2. A *Monitoring Service* that monitors objects and detects failures.
3. A *Consensus Service* used to implement group multicast and membership protocols.
4. A *Group Service* that provides both *group multicast* and *group membership*.

This architecture and the design of each of these services are described in Chapter 3. The dependencies between these components are *implementation-specific*, and do not appear in the IDL interfaces of the services. They are discussed in Chapter 4.

⋆
⋆ ⋆

Summary

The *Object Group Service* (OGS) is a service that provides group communication facilities in a CORBA environment. It has a modular, component-oriented architecture that complies with the CORBA design guidelines. All the OGS components are specified in terms of IDL interfaces, and are integrated together to provide group support. Individual components may be modified without the knowledge of the applications that use them, and they may be reused in different contexts unrelated with groups.

The OGS group model defines the notions of object group, group behavior, and group reference. These notions are different in the context of groups than with individual objects. Groups can be configured to encapsulate plurality and behavior, by giving the illusion to a client that it deals with a single object.

The aim of group transparency is to hide groups from the application. Transparency may be classified according to the point of view of the application that benefits from it (client-side, server-side, or member's group transparency), or according to its role in the architecture (plurality, type, or behavior transparency). Transparency is achieved using mechanisms that map a single request to multiple requests and multiple responses to a single response, that manage dynamically-typed requests, and that ensure system correctness by maintaining a consistent ordering of requests in the group. The main benefits from transparency are ease of use and reusability of existing code.

Like most group communication toolkits, the OGS components provide facilities for group multicast, group membership, agreement protocols, failure detection, and reliable messaging. These components are packaged as CORBA services, and are independent from each other; their dependencies only appear at the implementation level.

Chapter 3

The Object Group Service: Architecture

A complex system that works is invariably found to have evolved from a simple system that works.

J. Gall

This chapter presents the architecture and design of the different components of the OGS environment, using a top-down approach. Each of these components is described in isolation, and the sections that present them may be read independently. For each of them we discuss and justify the design choices we made, describe the semantics of the service, and present its IDL interfaces. The formalism used for representing the interfaces and their interactions is based on the OMT notation [RBP+91].

3.1 The Group Service

The *group service* is the core of the OGS environment. It is the service that actually provides object group support, and which the application programmer has to deal with. It implements two functionalities:

- *Group membership* manages the life cycle of object groups. It maintains the updated list of all correct group members. It provides support for joining and leaving groups, view change notification, and state transfer. A group membership service is generally associated with a failure detection mechanism for detecting group member failures.
- *Group multicast* provides support for sending multicast invocations to all the members of a group, with various reliability and ordering guarantees.

The group multicast and group membership services interact closely. In particular, multicast operations are defined on object groups rather than on sets of unrelated objects, thus involving group membership. Therefore, both services are contained in the set of interfaces forming the group service. Group membership and group multicast are presented separately in the next two sections, prior to the complete description of the group service.

3.1.1 Group Membership

The role of a group membership service is to manage memberships in a distributed system on behalf of the processes that compose it. OGS supports dynamic groups, i.e., the composition of the groups can change over time. New members can join an existing group, explicitly leave it, or may be implicitly removed from the group because of a failure. Objects that wish to join a group do so by contacting the membership service, which updates the list of group members. Once admitted to the group, an object may interact with other group members. Finally, if the object fails or leaves the group, the membership service will again update the list of group members. Dynamic group membership involves two kinds of protocols:

- A *view change protocol* is run each time the composition of a group changes. It ensures that every correct member of the group receives a *view change notification*, indicating the new composition of the group as a list of group members with mutually consistent rankings. View changes are totally ordered with each other.
- A *state transfer protocol* is an atomic operation that happens during view change, when a new member joins an existing group. It consists in obtaining the state from a current group member, and giving it to the new member. This protocol ensures that the state of all group members is kept consistent upon membership changes. The view change protocol can terminate only after a state transfer is completed.

The group membership interfaces define how OGS and group members interact. They are essentially composed of two types of objects (Figure 3.6): (1) service-specific *group administrator* objects that enable group members to change their status in the group (e.g., join and leave the group); and (2) application-specific *groupable* objects that enable OGS to call back to the application for view change and state transfer protocols. The `GroupAdministrator` interface is implemented by the service, and is used as a black box by the application. A group administrator is assigned to a single group at creation time, but there may be several group administrators on the same host or in the same process. The `Groupable` interface must be implemented by application objects that need to be members of a group. These interfaces are described in the rest of this section.

IDL Interfaces

```
1  // IDL
2  // The object is not member of the group.
3  exception NotMember {};
4  // The object is already member of the group.
5  exception AlreadyMember {};
6
7  // Composition of a group at a specific time.
8  struct GroupView {
9    // Group composition.
10   sequence<Groupable> composition_;
11   // View identifier.
12   unsigned long version_;
13 };
14
15 // Service's view of group members.
16 interface Groupable {
17   // Invoked upon view change.
18   void view_change(in GroupView view);
19   // Invoked upon state transfer on a current member.
20   any get_state();
21   // Invoked upon state transfer on a new member.
22   void set_state(in any state);
23 };
24
25 // Server-side group representative.
26 interface GroupAdministrator {
27   // Join the group.
28   void join_group(in Groupable member)
29     raises(AlreadyMember);
30   // Leave the group.
31   void leave_group(in Groupable member)
32     raises(NotMember);
33 };
```

Figure 3.1: IDL Interfaces for Group Membership

The group membership service defines interfaces and operations for dynamic group management and for state transfer (Figure 3.1). The group management operations include the ability of group members to join a group and to explicitly leave a group using the `join_group()` and `leave_group()` operations (lines 28–32) of group ad-

ministrators. In addition, OGS can notify the group members of a view change using the `view_change()` operation (line 18) of groupable objects.

The `GroupView` structure (lines 8–13) represents a stable view of a group at a given time. Group views have version numbers that are incremented every time the composition of the group changes. A group view is passed to the application upon view change.

The state transfer is defined by two operations on group members: `get_state()` and `set_state()` (lines 20–22). The state is transferred using a value of type `any`, which can contain any kind of application-specific data. In the current version of OGS, the state is transferred in a single message. This is inadequate if the data representing the state is large. In this situation, the interface should be modified so that state data is split into several parts which are sent separately. The `get_state()` operation would then return part of the state. The service could either invoke this operation successively on a current group member, or perform load sharing by invoking the operation on several group members. The `set_state()` operation would be then invoked multiple times on a new member.

Notice that the state transfer operations are necessary because the state of a group can contain implementation-specific data. If the state of the servers were only expressed in terms of IDL attributes accessible at runtime by the group service, the state transfer operations would be unnecessary.

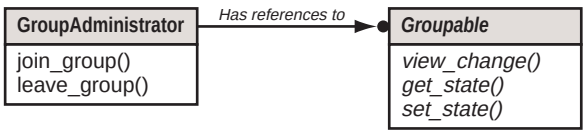

Figure 3.2: Class Diagram of Group Membership Interfaces

Interfaces and operations related to group membership are illustrated in the class diagram of Figure 3.2. Group administrators hold references to the groupable objects of a group, and invoke them asynchronously upon view change and state transfer. The `Groupable` interface is abstract (italicized in the figure), in the sense that the application has to inherit from it and implement its functionality.

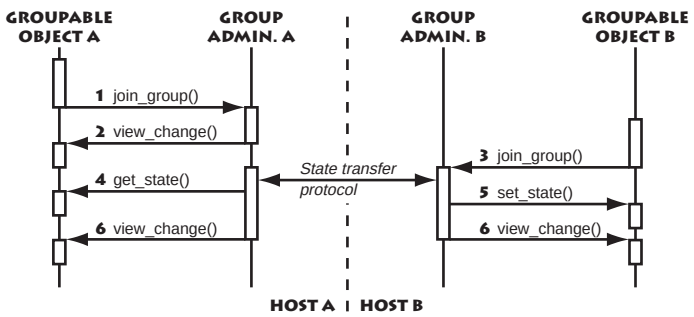

Figure 3.3: Interaction Diagram of View Change and State Transfer

The interaction diagram of Figure 3.3 illustrates the view change and state transfer mechanisms on two members. Initially, the group is empty. A first member object joins the group (1), and receives the new view (2). A second member object joins the group (3), which is not empty anymore, leading to both a state transfer protocol (4, 5), and a view change notification (6).

3.1.2 Group Multicast

Group multicast provides primitives for sending invocations to groups instead of singleton objects. OGS provides a rich set of group multicast primitives, adapted to various types of applications. OGS implements groups as open structures (see Section 1.2), and allows non-member objects to issue multicast invocations to groups. Multicast primitives can be classified according to their degree of *reliability* and their *ordering* guarantees. In addition, OGS provides two communication models: *untyped* and *typed* invocations.

Multicast Reliability

Reliable multicast is a key mechanism for developing replicated and parallel applications. In the context of group communication, it means that all correct group members deliver the same set of messages *(agreement)*, that this set includes all messages multicast to the group by correct objects *(validity)*, and that no spurious messages are ever delivered *(integrity)* [HT93].

OGS provides both unreliable and reliable multicast primitives. The unreliable primitives may be useful for read-only operations, but should be used with caution, since it may corrupt the state of the group if all messages are not received by every member. Reliable multicast, in itself, does not ensure that group consistency is preserved; it is generally combined with an ordering guarantee.

Multicast Ordering

In addition to reliability, message ordering is an important concern when dealing with one-to-many communication. The state of an object generally depends on the order in which it receives requests, as illustrated by Figure 5.7 in Chapter 5. There are various types of ordered multicasts:

- A *First-In First-Out (FIFO) multicast* guarantees that the reliable multicasts sent by the *same* object are delivered in the same order.
- A *totally ordered (atomic) multicast* guarantees that reliable multicasts are delivered in the same order to all target objects.
- A *causally ordered multicast* ensures that a message is not delivered until all the causally preceding messages have been delivered.

The current version of OGS provides unordered, FIFO, and totally ordered multicast primitives.

Multicast Replies

Just as an invocation to a singleton object may return a value, a multicast invocation may return a set of values (one from each target object). The client may want to wait only for the first reply, for several replies, or even for no replies. Since the composition of a group may change at any time, it is generally not wise to wait for an absolute number of replies; for instance, if the group contains n members, waiting for exactly n replies would lead the client to block if a member fails before having sent its reply. Therefore, in OGS, the number of replies expected by the client are defined in terms of values relative to the view composition:

- *All-replies multicast invocations* wait for replies from all group members.
- *Majority-of-replies multicast invocations* wait for a majority of replies from the group members.
- *One-reply multicast invocations* wait for the first reply from any group member.
- *Zero-reply (synchronous) multicast invocations* wait for the operation to complete on one group member, but does not return any reply to the client.
- *One-way multicast invocations* do not wait for any reply from the group members, and do not block the client's execution thread.

With either majority-of-replies or all-replies multicast invocations, OGS might need to modify the expected number of replies if a server crashes while the invocation is processed. The zero-reply synchronous invocation style is useful for flow control to avoid the object group becoming congested; it blocks the sender until the message is processed, allowing the receivers time to decongest.

Untyped Interfaces and Explicit Invocations

The untyped invocation interface of OGS allows us to send untyped messages to group members. An object that wishes to send a multicast to an object group must explicitly pack the data of the message into a value of type `any`, and pass it to OGS which will perform the multicast. On the server side, OGS delivers this message by passing the `any` value to group members through their `deliver()` operation.

This invocation interface is flexible since the client can place any kind of data in the message, and can easily specify the semantics associated with the multicast invocation and the expected number of replies. The drawback of this model is that the application programmer must *explicitly* pack all parameters associated with the server's invocation in a message (marshaling), and extract these parameters on the server side (unmarshaling), which is a painful and error-prone task. In addition, this

untyped invocation interface exposes group communication to the client application, while a common goal to most group-based systems is to hide groups from clients.

Typed Interfaces and Transparent Invocations

The typed invocation interface of OGS provides group transparency to clients, which can issue invocations to object groups as if they were invoking singleton objects. The client directly invokes operations of the server's interface using static stubs, and OGS delivers multicasts by directly invoking the relevant operation of the server, using static skeletons. OGS transparently filters messages and returns a single reply to the client. Of course, typed communication does require that all servers implement the same IDL interface.

Although less flexible than the untyped invocation interface, this model is much easier to use. The application developer does not need to perform the marshaling and unmarshaling of the request (these operations are performed transparently by OGS), and can benefit from the type safety of CORBA's static invocation interface. The client cannot specify the semantics to associate with a multicast invocation but, as we will see in the next section, this may be advantageously replaced by server-specified invocation semantics.

Server-Specified Invocation Semantics

As mentioned in Section 2.4, OGS provides server-specified invocation semantics: the server may associate specific semantics to each individual operation of its interface when using the typed version of OGS.[1] For instance, if an operation does not change the state of the server (read-only operation), the server may decide to deliver the invocation without ensuring total order. Since the server implements the operations, it knows their properties and the client does not need to be aware of them. Furthermore, the server can optimize client requests based on their semantics; for instance, two update operations do not have to be totally ordered with each other if they are commutative (e.g., they modify disjoint parts of the server's state). This approach is an improvement over the traditional model where a client asks for the strongest ordering guarantees for a message when it is unsure of the exact semantics of the associated operation.

IDL Interfaces

The interfaces and operations related to group multicast are shown in Figure 3.4. They introduce two types of objects: (1) service-specific *group accessor* objects that define how clients can issue multicast invocations through OGS; and (2) application-specific *invocable* objects used by OGS to deliver multicast invocations to the application.

[1] By default, OGS uses a total order multicast for invocations with unspecified semantics.

```
1  // IDL
2  // A sequence of anys for multicast return values.
3  typedef sequence<any> AnySeq;
4
5  // Number of replies to wait for.
6  enum NumReplies { ONEWAY, ZERO, ONE, MAJORITY, ALL };
7  // Request semantics (not an enum for extensibility).
8  typedef short Semantics;
9  const Semantics UNRELIABLE  = 0;
10 const Semantics RELIABLE    = 1;
11 const Semantics FIFO        = 2;
12 const Semantics TOTAL_ORDER = 3;
13
14 // Service's view of multicast target objects.
15 interface Invocable {
16   // Invoked upon message delivery.
17   any deliver(in any msg);
18 };
19
20 // Client-side group representative.
21 interface GroupAccessor {
22   // Issue a multicast to the group.
23   AnySeq multicast(in any msg,
24                    in NumReplies replies,
25                    in Semantics sem);
26   // Return a typed group accessor.
27   Object cast(in CORBA::InterfaceDef id);
28 };
```

Figure 3.4: IDL Interfaces for Group Multicast

Group accessors are the client-side equivalent of group administrators. They are used by clients to multicast data (contained in a value of type `any`) to specific groups, using the `multicast()` operation (lines 23–25). This operation allows us to specify the number of expected replies and the semantics associated with the invocation. These semantics are not defined as an enumeration, but as an integer value, so that implementations of OGS can add and support new types of semantics without modifying the interface definitions.

Each group accessor is associated with a single group at creation time. The `cast()`[2] operation (line 27) may be used to obtain a reference to a typed group accessor that supports the same interface as the group members, for transparent group invocation. Once a message has been multicast to a group, OGS invokes the `deliver()` operation (line 17) of each group member, defined in the abstract `Invocable` interface. Figure 3.5 illustrates these interfaces and operations in a class diagram.

3.1.3 The Complete Group Service

In the previous sections, we have introduced group membership and group multicast as two separate components. In OGS, both problems are addressed by the same

[2] `cast()` has no relationship with `multicast()`, although both operation names sound similar. Its name comes from *type* cast, which transforms one type to another.

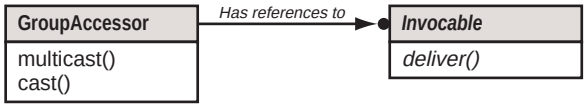

Figure 3.5: Class Diagram of Group Multicast Interfaces

service, although they are semantically distinct. This is due to the fact that both components have to interact very closely at the architectural level. This section presents the complete OGS specification, which comprises and extends the interfaces previously introduced.

OGS Components and Interactions

OGS combines support for group membership (see Section 3.1.1) and group multicast (see Section 3.1.2) in a single set of interfaces. Figure 3.6 presents a simplified high-level view of OGS components. OGS interfaces are classified according to three categories, associated with the different views of the service:

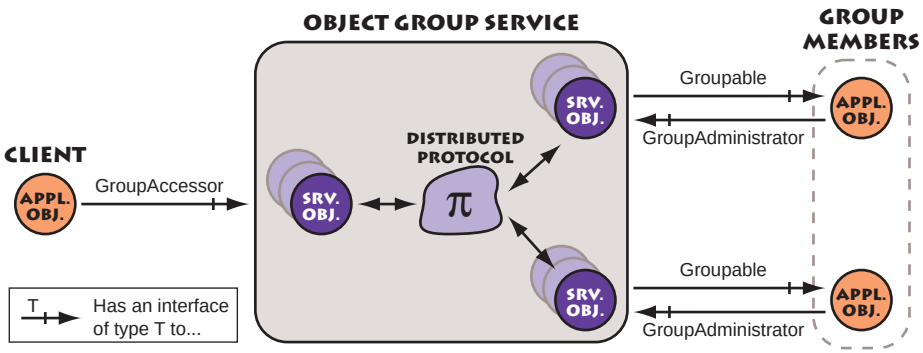

Figure 3.6: OGS Components Overview

1. *Client interfaces* (`GroupAccessor`) allow clients to interact with object groups.
2. *Member interfaces* (`GroupAdministrator`) are a superset of client interfaces, and allow servers to manage the group's life cycle (e.g., join and leave groups).
3. *Service interfaces* (`Groupable`) define interfaces that the member objects must implement for OGS to issue callbacks to them.

Group accessors and administrators are service objects. Performing a multicast to the group initiates a distributed protocol between group accessors and administrators, which ensures that messages are delivered to the members according to some condition (e.g., total order). The `Groupable` interface must be implemented by application objects.

OGS Group References

In OGS, group accessors and administrators act as *group references* (see Section 2.3). Similarly to an object reference, a client that holds one of these objects can issue invocations to the group designated by the reference, without having to know the location or identity of the objects that compose the group. However, unlike object references, a group reference is a *pure CORBA object*.

The drawback of this approach is that group references are created and managed explicitly by the application, whereas object references are created and managed by the ORB. In OGS, the application deals with object references pointing to group references (group accessors and administrators), leading to an extra indirection. In addition, it is currently not possible to pass a group reference *by value*[3] to another object.

On the other hand, considering group references as values would require the ORB to know about object groups and thus to extend the core CORBA specification, which is outside the scope of this thesis. Adding group support in the ORB core would also overload it with complex mechanisms that are only necessary for a limited set of applications. In addition, group references are semantically different from object references: an object reference reliably designates a unique object, while group references designate a set of objects whose membership changes over time, and that may even be empty. Group references are thus not self-contained; they require an intermediary agent to determine the group's current composition.

OGS IDL Interfaces

The full set of OGS interfaces, with their inheritance relationships, are presented in Figures 3.7 and 3.8. OGS interfaces are defined in two modules: the first one, `mGroupAccess`, is used by group clients and defines operations to communicate with object groups; the second module, `mGroupAdmin`, is used by group members and defines operations for administrating object groups. Interfaces from the `mGroupAdmin` module inherit from interfaces defined in `mGroupAccess`. Inheritance is used because the client depends on less functionality than the full interface offers (see section 2.2.1).

The `mGroupAccess` Module. The `Invocable` interface (lines 32–35) defines a limited view of groupable objects for the clients. A group view may be obtained from the group accessors, and is composed of a list of invocable objects. It differs from the view delivered to group members, in that there is no guarantee that this view is fully up-to-date on the client side. Although clients may directly invoke individual group members using the `deliver()` operation (line 34) of the `Invocable` interface, this facility must be used with care since group consistency will be broken by modifying the state of individual group members.

A group accessor is created using an object factory (lines 56–60), which permits the

[3] The next CORBA revision will support passing objects by value.

```
1  // IDL
2  // Client's view of OGS.
3  module mGroupAccess {
4    // A sequence of anys for multicast return values.
5    typedef sequence<any> AnySeq;
6
7    // Number of replies to wait for.
8    enum NumReplies { ONEWAY, ZERO, ONE, MAJORITY, ALL };
9    // Request semantics (not an enum for extensibility).
10   typedef short Semantics;
11   const Semantics UNRELIABLE  = 0;
12   const Semantics RELIABLE    = 1;
13   const Semantics FIFO        = 2;
14   const Semantics TOTAL_ORDER = 3;
15
16   // Forward reference.
17   interface Groupable;
18
19   // Composition of a group at a specific time.
20   struct GroupView {
21     // Group composition.
22     sequence<Groupable> composition_;
23     // View identifier.
24     unsigned long version_;
25   };
26
27   // Error while performing an operation on the group.
28   exception GroupError { string description_; };
29   // Error while performing an operation on a non-existent group.
30   exception NoGroup {};
31   // Invalid group name (e.g., containing invalid characters).
32   exception InvalidGroupName {};
33
34   // Client's view of group members.
35   interface Invocable {
36     // Invoked upon message delivery.
37     any deliver(in any msg);
38   };
39
40   // Client-side group representative.
41   interface GroupAccessor {
42     // Issue a multicast to the group.
43     AnySeq multicast(in any msg,
44                      in NumReplies replies,
45                      in Semantics sem)
46       raises(GroupError);
47     // Return a typed group accessor.
48     Object cast(in CORBA::InterfaceDef id)
49       raises(GroupError);
50     // Get the latest group view (possibly not up-to-date).
51     GroupView get_view()
52       raises(GroupError);
53     // Destroy the group accessor.
54     void destroy()
55       raises(GroupError);
56   };
57
58   // Factory for creating group accessors.
59   interface GroupAccessorFactory {
60     // Create a group accessor.
61     GroupAccessor create(in string group_name)
62       raises(GroupError, NoGroup, InvalidGroupName);
63   };
64 };
```

Figure 3.7: IDL Interfaces of the `mGroupAccess` Module

creation of objects in a remote address space. The destruction of a group accessor is performed using its `destroy()` operation (lines 51–52). Each group accessor is associated to a single group at creation time. This association is performed by passing a group name as parameter of the `create()` operation (lines 58–59). A group name is a system-wide unique identifier that can have the form of a *Uniform Resource Locator* (URL), such as `ogs://banks.ch/accounts/John.Smith`.

```
1  // IDL
2  // Server's and service's view of OGS.
3  module mGroupAdmin {
4    // The object is not member of the group.
5    exception NotMember {};
6    // The object is already member of the group.
7    exception AlreadyMember {};
8
9    // Semantics associated to an operation.
10   struct OperationSemantics {
11     // Operation name.
12     CORBA::Identifier name_;
13     // Operation semantics.
14     mGroupAccess::Semantics semantics_;
15   };
16   // Set of operations commutative with each other.
17   typedef sequence<CORBA::Identifier> CommutativeOperations;
18   // Semantics associated to the operations of an interface.
19   struct InterfaceSemantics {
20     // Default semantics for operations.
21     mGroupAccess::Semantics default_semantics_;
22     // Semantics associated to specific operations.
23     sequence<OperationSemantics> operation_semantics_;
24     // Sets of operations commutative with each other.
25     sequence<CommutativeOperations> commutative_operations_;
26   };
27
28   // Service's view of group members.
29   interface Groupable : mGroupAccess::Invocable {
30     // Invoked upon view change.
31     void view_change(in mGroupAccess::GroupView view);
32     // Invoked upon state transfer on a current member.
33     any get_state();
34     // Invoked upon state transfer on a new member.
35     void set_state(in any state);
36   };
37
38   // Server-side group representative.
39   interface GroupAdministrator : mGroupAccess::GroupAccessor {
40     // Join the group.
41     void join_group(in Groupable member,
42                     in InterfaceSemantics semantics)
43       raises(mGroupAccess::GroupError, AlreadyMember);
44     // Leave the group.
45     void leave_group(in Groupable member)
46       raises(mGroupAccess::GroupError, NotMember);
47   };
48
49   // Factory for creating group administrators.
50   interface GroupAdministratorFactory {
51     // Create a group administrator.
52     GroupAdministrator create(in string group_name)
53       raises(mGroupAccess::GroupError, mGroupAccess::InvalidGroupName);
54   };
55 };
```

Figure 3.8: IDL Interfaces of the `mGroupAdmin` Module

The `mGroupAdmin` Module. The `mGroupAdmin` module is used only by group members and defines operations for administrating object groups. Group members can issue multicasts into their own groups, and thus the `GroupAdministrator` interface (lines 39–50) inherits from the `GroupAccessor` interface defined in the `mGroupAccess` module.

When joining a group, a member object may specify the semantics associated with each of its operations. To do this, it must pass a structure of type `InterfaceSemantics` (lines 10–26) to the `join_group()` operation (lines 41–43). This structure includes the default semantics for the operations of the group member (line 21), a list of operations with their associated semantics (line 23), and a list containing sets of commutative operations (line 25) that may be used by the implementation to optimize protocols when ordering concurrent requests.

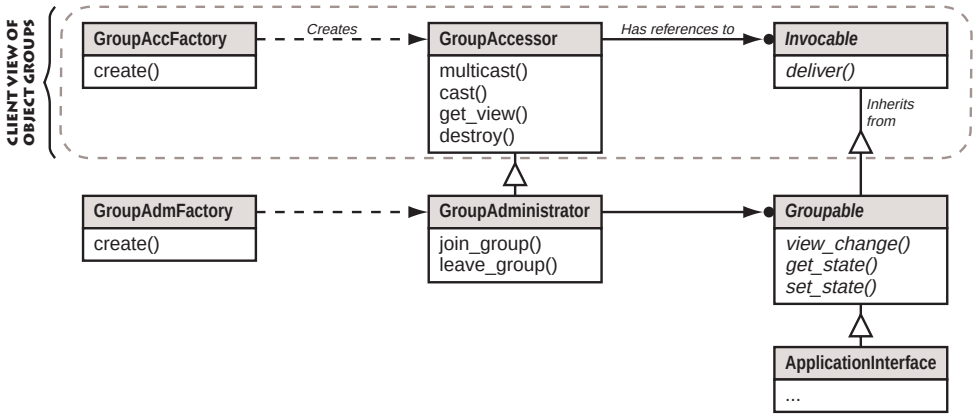

Figure 3.9: Class Diagram of OGS Interfaces

Figure 3.9 illustrates the interfaces and operations of the complete OGS specification in a class diagram. Inheritance relationships clearly show that server-side interfaces inherit from client-side interfaces. An application-specific group member must inherit from the abstract `Groupable` interface.

3.1.4 OGS Extensions for Replication Support

Group communication is well adapted to object replication. Nevertheless, not all replication models can be used transparently with the group interfaces previously described. This section gives an overview of the architectural differences between the active and primary-backup replication models, discusses the different approaches for dealing with these replication models, and presents the required extensions to the group service interfaces.

OGS and Active Replication

The active replication model is symmetrical: all group members have the same behavior, which is to accept a request, handle it, and optionally return a reply. It is easy to actively replicate an existing application (assuming that it has deterministic behavior) without having to re-implement its operations, by simply inheriting from the OGS `Groupable` interface. For the code that implements application-specific operations, replication is *transparent*: OGS directly invokes the target operation and request processing is performed the same way whether the object is replicated or not. Therefore, no extra support is required from the group service for implementing actively replicated servers.

OGS and Primary-Backup Replication

Primary-backup replication distinguishes between the primary object, which processes the requests, and the replicas, which only receive updates from the primary. The update mechanism handles the reply sent to the client and the update information sent to the backups quite differently. The latter contains the modifications that the processing of the request has induced in the state of the primary, while the former may contain completely unrelated data.

These different roles of primaries and backups require some support interfaces to be added to the group service. This support may be implemented in different ways:

1. Limit the use of primary-backup replication to the untyped version of OGS, and adapt the OGS interfaces to deal with transmission of updates from the primary to the backups.

2. Restrict primary-backup replication to untyped communication as well, but without modification to the OGS interfaces. Since the `deliver()` operation of groupable objects returns a value of type `any`, the primary server could return both the reply and the update information in this value using a well-specified format. Similarly, the `any` value passed to `deliver()` could contain a regular message *or* an update from the primary, and could be checked at runtime using the type-safe extraction mechanisms of CORBA. OGS would update the backups using these properties.

3. Enforce the programmer of the primary server to explicitly deal with primary-backup replication in his implementation. This approach has been adopted by Orbix+Isis in their so-called coordinator-cohort model: in the implementation of each operation, the application has to call a function that indicates whether the server is the primary or a backup, and must process the request accordingly; the update is transmitted by calling another Orbix+Isis-specific function. The drawback of this approach is that there is no group member's transparency (see Section 2.4), and it makes it impossible to replicate existing applications without extensive modifications of the implementation.

4. Use the state transfer mechanism for updates. If the state is very small (e.g., a single value), it may seem reasonable to use the state transfer mechanism to update the backups in a transparent way after processing each request. This approach is of course not viable if the state is large, and it tends to generate unnecessary state transfers for operations that do not modify the state.

In OGS, we have adopted the first approach. We have subclassed the interfaces of the group service to deal with primary-backup replication. The resulting module, shown in Figure 3.10, defines a new type of communication semantics: `PRIMARY_BACKUP` (line 5). This constant may be given to an OGS implementation that supports primary-backup replication when multicasting a message.

```
1  // IDL
2  // Primary-backup support for OGS.
3  module mPrimaryBackup {
4    // New semantics for primary-backup communication.
5    const mGroupAccess::Semantics PRIMARY_BACKUP = 4;
6
7    // Structure to hold the reply and update information.
8    struct ReplyAndUpdate {
9      // Reply for the client.
10     any reply_;
11     // Update for the backups.
12     any update_;
13   };
14
15   // Service's view of group members.
16   interface PrimaryBackupGroupable : mGroupAdmin::Groupable {
17     // Invoked upon primary message delivery.
18     ReplyAndUpdate deliver_primary(in any msg);
19     // Invoked upon primary message delivery.
20     void update_backup(in any update);
21   };
22 };
```

Figure 3.10: IDL Interfaces for Primary-Backup Replication

To deal with the transmission of updates, we have defined a structure, `ReplyAndUpdate` (lines 8–13), that contains both the reply for the client and the update information for the backups. This structure is returned by the `deliver_primary()` operation (line 18), invoked on the primary copy. The backups receive the update information through their `update_backup()` operation (line 20). Objects that support primary-backup replication must implement the `PrimaryBackupGroupable` interface (lines 16–21), which inherits from `Groupable`. In doing this, a member object may use active *and* primary-backup replication at the same time with a group service implementation that supports both models. Notice that group member's transparency is not possible with primary-backup replication because the server objects have to implement explicit update support for each of their operations.

3.1.5 Applicability of the Service

The interfaces of the group service are designed for group communication in general, and can thus be used in contexts where the notion of object groups is a suitable paradigm. In particular, application domains such as high availability, fault tolerance, parallel processing, or collaborative work make extensive use of object groups. Examples of applications that use OGS are given in Chapter 5.

In addition, the group service may be used as the base for other services. For instance, a replication service could provide high-level facilities for replicating objects without exposing groups to the application. Such a service could use the group service for implementing replication support.

The group service could also be used for fault tolerant and highly available implementations of CORBA services, such as replicated event channels or a fault tolerant naming service. A similar approach has been adopted by Electra for the implementation of its fault tolerant naming service [Maf96].

⋆
⋆ ⋆

3.2 The Consensus Service

The *Object Consensus Service* is a CORBA service that allows a set of application objects to solve the so-called distributed consensus problem [BMD93]. This service is a generic building block, useful for all kinds of applications where distributed agreement problems have to be solved. In this section, we explore the distributed consensus problem and present how it can be expressed in terms of interactions between distributed components. We then describe the design of the CORBA consensus service and its IDL interfaces. This service is architecturally unrelated to the group service interfaces presented in the previous section, although dependencies exist at the implementation level. These dependencies are described in the section dedicated to the OGS implementation, together with the protocol for solving the consensus problem.

3.2.1 The Consensus Problem

Informally, a consensus allows several processing elements to reach a common decision, according to their initial values, despite the crash of some of them [BMD93]. The consensus problem is a central abstraction for solving various agreement problems (atomic commitment, total order, membership) [GS96], and thus for achieving fault tolerance in distributed systems. Agreement problems are present in many distributed systems, such as systems based on virtual synchrony, transactions, or replication, and are generally implemented using ad hoc protocols.

The consensus problem can be defined in terms of two primitives: $propose(v)$ and $decide(v)$, where v is a value. All correct participants propose an initial value, and must agree on some final value related to the proposed values. The consensus problem between a set of processes is specified as follows [CT96]:

- *Termination:* every correct process eventually decides some value.
- *Uniform integrity:* every process decides at most once.
- *Agreement:* no two correct processes decide differently.
- *Uniform validity:* if a process decides v, then v was proposed by some process.

Instead of using ad hoc solutions for implementing the agreement protocols necessary to implement group communication, we propose a generic consensus service that may be used to solve these various agreement protocols. In addition to its modularity, this approach enables efficient implementations of the protocols as well as precise characterization of their liveness [GS96]. Consensus is provided at the object level rather than at the process level. The consensus service is entirely defined in terms of IDL interfaces, and is usable between heterogeneous objects.

3.2.2 Architecture

Consensus Components and Interactions

The consensus service is essentially composed of two categories of objects (Figure 3.11):

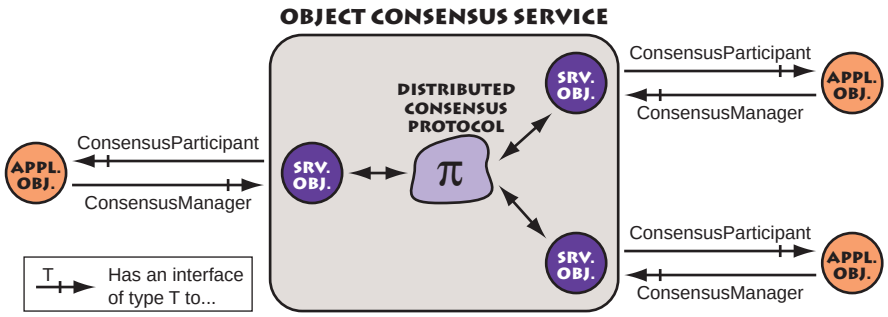

Figure 3.11: Object Consensus Service Components Overview

- *Consensus managers* are service-specific objects that implement the consensus protocol and reach agreement with each other. The consensus manager acts as a black box, and its implementation is provided by the service. The underlying consensus protocol is not exposed to the application and may be changed without impact on the clients of the service.
- *Consensus participants* are application-specific objects that are only involved in the consensus for proposing an initial value and receiving the decision. They are implemented by the application and allow the customization of the consensus for particular tasks. They provide the generic dimension of the consensus.

Consensus managers and participants interact in two situations: (1) Initially, consensus participants propose their estimate to consensus managers; and (2) once a decision has been taken, the consensus managers deliver it to the consensus participants.

Decision delivery is a typical situation where a callback interface (see Section 2.2.1) is useful. The service can asynchronously notify the application that the consensus has been solved and that a decision value is available. Without a callback interface, the client would be blocked during the consensus execution, or would need to poll for the decision value. In addition, the callback interface allows us to clearly define how the application participates in the service. The *proposition of an initial value* impacts on the consensus instantiation model, and is discussed in next section.

Consensus Instantiation

When initiating a consensus, participants must give their initial value to consensus managers. For doing this, two alternatives are available: (1) either the participant

gives its initial value as a parameter to the operation that launches the consensus, or (2) the consensus manager calls back to the participant to obtain this value.

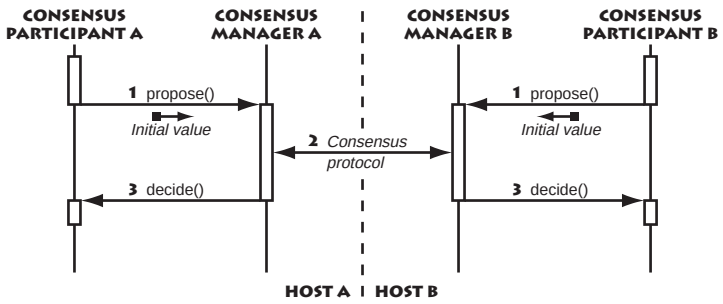

Figure 3.12: Explicit Consensus Instantiation

The first alternative has the advantage in that it conforms exactly to the model of Chandra and Toueg's consensus algorithm used in OGS (see Section 4.3.2). The participant gives its initial value to the consensus manager as parameter of the operation that initiates a consensus. This model *requires* that *every* participant explicitly invokes this operation on the consensus manager in order to launch the consensus protocol (explicit instantiation). Figure 3.12 illustrates this mechanism: each participant initiates a consensus by invoking `propose()` on a consensus manager (1), which launches the consensus protocol (2) and delivers the decision to the participant (3).

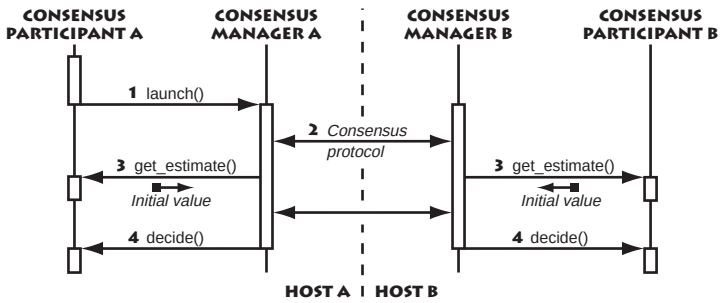

Figure 3.13: Implicit Consensus Instantiation

The second alternative consists in having the consensus managers asynchronously obtain the initial value from the participants, after the consensus has been launched. It is a more general model since all participants do not need to explicitly launch the consensus for the consensus protocol to be run. Objects may participate *passively* in the consensus protocol (implicit instantiation). Notice that implicit instantiation requires a participant to be attached to each consensus manager prior to consensus instantiation. Figure 3.13 illustrates this mechanism: at least one participant initiates a consensus by invoking `launch()` on a consensus manager (1), which starts the consensus protocol (2); each manager gets the initial value from its participant (3), executes the consensus protocol, and finally delivers the decision to the

participant (4).

The choice of the model to adopt can thus be summarized in one question: *do all participants need to explicitly instantiate a consensus for the protocol to be run?* We believe that the choice depends on the application and the consensus algorithm, and that this decision should appear at implementation level, not in the interface. Therefore, we adopted the second alternative in OGS since it can be used for both implicit and explicit consensus instantiation, and thus provides for more implementation flexibility.

3.2.3 Design

In addition to the design goals common to all CORBA services, such as simplicity, orthogonality, flexibility, genericity, and reusability, other goals have guided the design of the consensus service:

- A consensus is generic; it decides on application-specific data.
- The protocol used for solving the consensus is implementation-specific, and should not be exposed to the application.
- The instantiation model of the consensus is an implementation characteristic (see Section 3.2.2).
- Consensus instances are independent; two instances may involve distinct sets of consensus managers and participants.
- Several consensus instances may execute in parallel, if the implementation supports it.
- Efficiency is a major concern; interfaces must ensure that efficient implementations are possible.

The interfaces of the object consensus service take these design goals and principles into account. In particular, as discussed below, some design choices have been made to maximize implementation efficiency.

Service Interfaces

The interfaces of the consensus service are presented in Figure 3.14. The service uses untyped `any` variables for proposition and decision values, so that a consensus can decide on application-specific data. Before using the consensus service, an application must create a `ConsensusManager` object (lines 28–36). This is performed using a consensus manager factory (lines 39–42). Upon creation, each consensus manager is associated for its entire lifetime with a `ConsensusParticipant` object (lines 19–25). Several consensus managers can be located in the same process or on the same host, and a participant may be associated with several consensus managers.

```
1  // IDL
2  module mConsensus {
3    // Forward declaration.
4    interface ConsensusManager;
5
6    // A consensus identifier.
7    typedef long ConsensusId;
8
9    // A sequence of consensus managers.
10   typedef sequence<ConsensusManager> ConsensusManagerSeq;
11
12   // Object that represents a set of consensus managers.
13   interface ConsensusView {
14     // Destroy a consensus view.
15     void destroy();
16   };
17
18   // Object that can participate to a consensus.
19   interface ConsensusParticipant {
20     // Return the participant's estimate.
21     any get_estimate(in ConsensusId cid);
22     // Give the decision to the participant.
23     void decide(in ConsensusId cid,
24                 in any decision);
25   };
26
27   // Object that can launch and execute a consensus.
28   interface ConsensusManager {
29     // Create a new consensus view.
30     ConsensusView create_view(in ConsensusManagerSeq cms);
31     // Launch a new consensus and return.
32     oneway void launch(in ConsensusId cid,
33                        in ConsensusView view);
34     // Destroy the consensus manager.
35     void destroy();
36   };
37
38   // Factory for creating consensus managers.
39   interface ConsensusManagerFactory {
40     // Create a consensus manager.
41     ConsensusManager create(in ConsensusParticipant cp);
42   };
43 };
```

Figure 3.14: IDL Interfaces of the Object Consensus Service

An application can launch a consensus by invoking the `launch()` operation (lines 32–33) of a consensus manager. This operation expects a unique identifier and a consensus view — i.e., the list of participating consensus managers — as parameters. Once a consensus has been launched, each manager gets the estimate of its associated participant by invoking its `get_estimate()` operation (line 21). Then, the consensus protocol is executed between all participating consensus managers. When the decision is reached, the consensus managers give it to the participants through their `decide()` operation (lines 23–24). Since the estimate is obtained asynchronously from the participant, it is possible for a third party to launch a consensus protocol, while ensuring that the estimate is actually proposed by the participant.

Each independent consensus can be run with a different set of participants. This set is specified through a parameter of type `ConsensusView` (lines 13–16) given to the consensus manager upon consensus initiation. A consensus view is defined as an object rather than as a structure for efficiency reasons. Indeed, in our implementation of the consensus service, consensus managers use the monitoring service (see Section 3.3) to monitor each other. If a consensus view were defined as a structure, consensus managers would have to be added to the list of monitored objects each time a new consensus is initiated, and removed upon its termination. These operations are costly in terms of performance, and are unnecessary when several consensus instances are launched with the same set of participants. Using an object as a consensus view gives a chance to perform these addition and removal operations only once upon object creation and deletion.

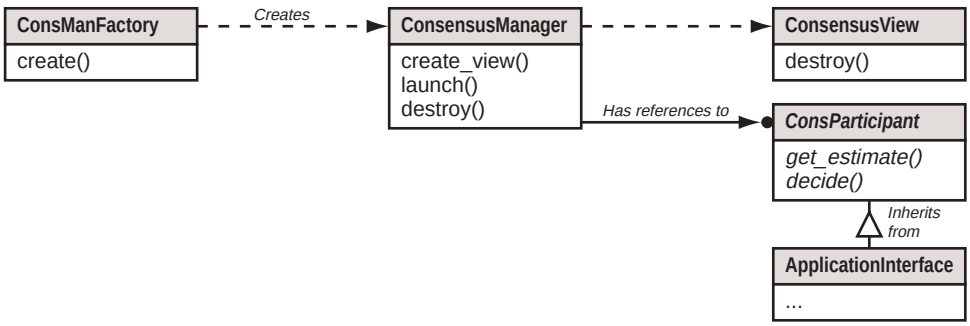

Figure 3.15: Class Diagram of the Object Consensus Service Interfaces

Figure 3.15 illustrates the interfaces and operations of the complete consensus service specification in a class diagram. Applications adapt the generic consensus protocol to their needs by inheriting from the abstract `ConsensusParticipant` interface.

3.2.4 Applicability of the Service

The consensus service may be used by any distributed system that needs to conciliate or synchronize its components. Application domains that benefit from distributed agreement are numerous. For instance, the consensus service may be used for allocating distributed resources (e.g., printers) or for synchronizing parallel tasks.

In addition, the consensus service may be used at a lower level for the implementation of *distributed protocols*, such as atomic commitment, total order, and membership. It constitutes thus a basic building block for distributed protocol support.

⋆
⋆ ⋆

3.3 The Monitoring Service

The *Object Monitoring Service* is a service that provides failure detection mechanisms in CORBA. It can be used by any application that requires knowledge about remote component failures. It targets large systems where thousands of objects are monitored, and allows a reduction of remote invocations between failure detectors and the objects being monitored. In the following, the terms *monitoring* and *failure detection* are considered equivalent.

In this section, we present the failure detection problem and architectural solutions to address this problem. We do not focus only on a single architecture; instead, we present and compare several different architectural approaches to provide object monitoring, and we show how they can be expressed in terms of IDL interfaces. We point out the advantages and disadvantages of each, and finally we end up by describing the generic monitoring service that was adopted for the OGS environment.

3.3.1 The Failure Detection Problem

Many of today's distributed applications have requirements for high availability and fault tolerance. These applications are based on protocols that take into account component failures. Unfortunately, agreement problems such as distributed consensus, atomic commitment, total order, and group membership, are not solvable with deterministic algorithms in an asynchronous system if one single process may crash [FLP85]. This impossibility is caused by the difficulty of distinguishing a "very slow" process from a crashed one.

Chandra and Toueg propose a way to circumvent this impossibility [CT96], by augmenting asynchronous systems with a failure detection mechanism that can make mistakes. In particular, they introduce the concept of *unreliable failure detectors* for systems with *crash* failures. The underlying idea is to reduce the asynchrony of the system by augmenting it with knowledge about component failures (which may be imperfect), making it possible to solve many problems that are otherwise unsolvable.

A *Failure Detector* (FD) is a service, usually local to each process, that provides information about component failures. It monitors a subset of the components in the system, and maintains a list of those it currently suspects to have crashed. In order to solve agreement problems, a failure detector must satisfy the following properties in an asynchronous system with a majority of correct processes[4] [CHT96]:

- *Eventual weak completeness:* there is a time after which every process that crashes is always suspected by some correct process.
- *Eventual weak accuracy:* there is a time after which some correct process is never suspected by any correct process.

[4]The standard formalism for describing concepts and algorithms in distributed systems specifies interactions between *processes*. When applying these concepts to OGS, we consider interactions between *objects* or *components*.

While the completeness property is not difficult to achieve, no algorithm can satisfy eventual weak accuracy in an asynchronous system. In most practical situations, however, it is sufficient if a failure detector satisfies this property "long enough" [GS97].

Failure detector implementations are generally based on timeout mechanisms. The choice of timeout values is crucial for the failure detector's ability to respect accuracy and completeness. Short timeouts allow a process to detect failures quickly but increase the number of false suspicions, with a risk of violating accuracy. Hence, there is a trade-off between latency (short timeouts) and accuracy (long timeouts). As soon as a system involves more than one *Local Area Network* (LAN), the optimal timeout value between two objects depends on both their respective locations in the system and on the characteristics of the underlying network.

3.3.2 Architecture

Monitoring Components and Interactions

A system that provides monitoring facilities involves different kinds of objects. Clients use the monitoring facilities provided by these objects through well-defined interfaces. In addition to clients, there are generally three categories of objects in a monitoring system:

- *Monitors (or failure detectors)* are the objects that collect information about component failures. In many systems, monitoring mechanisms are directly implemented in the client's code, and they do not appear as separate objects.
- *Monitorable objects* are objects that may be monitored, i.e., the failure of which may be detected by the failure detection system.
- *Notifiable objects* are objects that can be registered by the monitoring service, and that are asynchronously notified about object failures. Not all monitoring systems provide asynchronous failure notifications, and thus do not provide notifiable objects.

Monitorable and notifiable objects are generally application-specific, while monitors are implemented by the service. There are two types of interactions between these components:

- *Monitor ↔ client, monitor ↔ notifiable object:* this interaction allows the application to obtain information about component failures.
- *Monitor ↔ monitorable objects:* this interaction is performed by the monitoring service to continuously keep track of the status of monitorable objects.

Figure 3.16 illustrates the components and interactions of an object monitoring system. This sample configuration comprises a client, a notifiable object, a monitor,

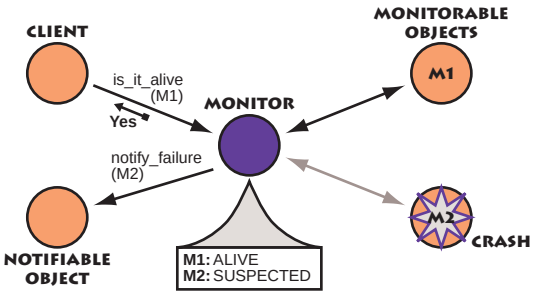

Figure 3.16: Components and Interactions of an Object Monitoring System

and two monitorable objects $M1$ and $M2$. The monitor keeps track of component failures. The client explicitly asks the monitor about the status of monitorable objects. Upon the crash of a monitored object, the monitor asynchronously informs the notifiable object of the failure.

Monitoring Models

Several models can be used for object monitoring. These models differ depending on the way the information about component failures is propagated in the system, i.e., the *flow policy*. There are two basic forms of unidirectional flow, *push* and *pull*, plus several variants. These flow policies are outlined in the rest of this section.

Push Model. In the *push* model, the direction of control flow matches the direction of information flow. With this model, monitored objects are active. They periodically send *heartbeat* messages to inform other objects that they are still alive. If a failure detector does not receive the heartbeat from a monitored object within specific time bounds, it starts suspecting the object. This method is efficient since only *one-way* messages are sent in the system, and it may be implemented with hardware multicast facilities if several failure detectors are monitoring the same objects.

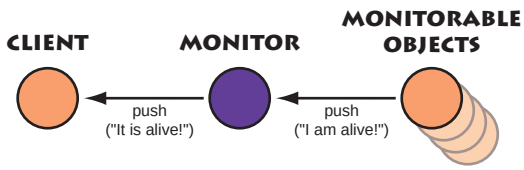

Figure 3.17: The Push Model for Object Monitoring

Figure 3.17 illustrates how the push model is used for monitoring objects. Notice that the messages exchanged between the monitor and the client are different from the heartbeat messages sent by monitored objects. The monitor generally notifies the client *only* when a monitored object changes its status (i.e., becomes suspected

or is no longer suspected), while heartbeat messages are sent continuously.

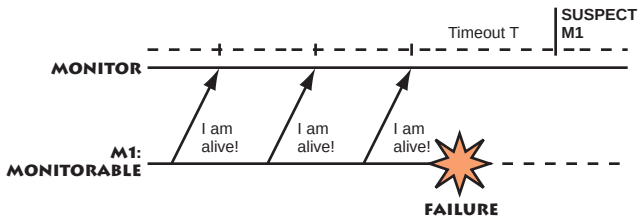

Figure 3.18: Monitoring Messages with the Push Model

The messages exchanged between the monitor and the monitorable object with the push model are shown in Figure 3.18. The monitorable object periodically sends heartbeat messages to the monitor. Upon message reception, the monitor sets a timer that triggers a suspicion if it expires before the reception of a new heartbeat message from the same object.

Pull Model. In the *pull* model, information flows in the opposite direction of control flow, i.e., only when requested by consumers. With this model, monitored objects are passive. The failure detectors periodically send *liveness requests* to monitored objects. If a monitored object replies, it means that it is alive. This model may be less efficient than the push model since *two-way* messages are sent to monitored objects, but it is easier to use for the application developer since the monitored objects are passive, and do not need to have any time knowledge (i.e., they do not have to know the frequency at which the failure detector expects to receive messages). Figure 3.19 illustrates how the pull model is used for monitoring objects.

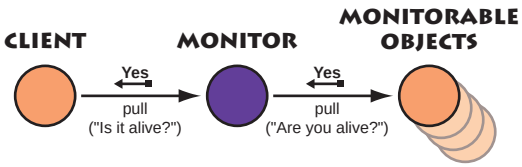

Figure 3.19: The Pull Model for Object Monitoring

The messages exchanged between the monitor and the monitorable object with the pull model are shown in Figure 3.20. The monitor sends periodically a liveness request to the monitorable objects, and waits for a reply. If it does not get the reply, a timeout triggers a suspicion.

Dual Model. To provide more flexibility to the monitoring system, we introduce an extension of the previous models, called the *dual* model, in which both the push and pull models can be used at the same time with the same set of objects. The basic idea is to use two kinds of *one-way* messages for monitoring objects. The failure detector sends a *one-way liveness request* to a monitorable object (pull model), and

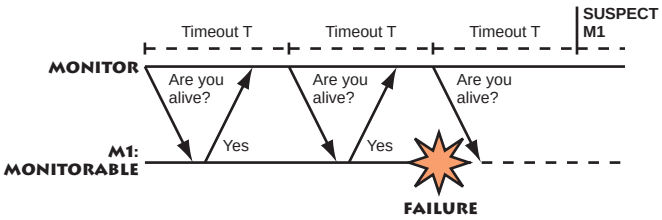

Figure 3.20: Monitoring Messages with the Pull Model

the latter replies with a *one-way liveness message* (push model). The monitorable object may also send a liveness message without having been requested to; in this case, we have a plain push model. The failure detector sends a liveness request only if the monitorable did not send a liveness message within some specific time bounds. This model works with any type of monitorable objects, without requiring the monitor to know which model is supported by every single object.

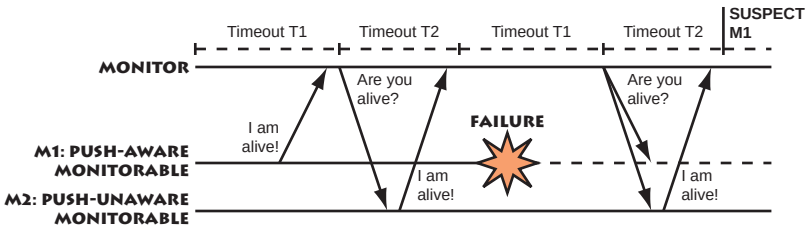

Figure 3.21: Monitoring Messages with the Dual Model

Figure 3.21 illustrates object monitoring with the dual model. In this example, two objects are being monitored. The first object, $M1$ is push-aware, i.e., it is active and periodically sends liveness messages (heartbeats). The second one, $M2$, is not push-aware, i.e., it only sends liveness messages when it is asked to. The monitor uses two timeout periods $T1$ and $T2$. It expects liveness messages of push-aware monitorable objects during $T1$. After $T1$, the monitor sends a liveness request to each monitorable object from which it did not receive a liveness message, expecting a reply during $T2$. After $T2$, the monitor suspects every process from which it did not receive a message. In this example, $M1$ sends a liveness message during $T1$ in the first phase, and crashes soon after. In the second phase, the monitor sends a liveness request to $M1$, but does not get a liveness message before the end of $T2$. Thus, it starts suspecting $M1$ to have crashed.

Evaluation of the Monitoring models. The interaction between monitors and monitorable objects seems to favor pull communication: with the pull model, the failure detector parameters (e.g., timeouts, which may need dynamical adjustment) need only reside in the failure detector and are not distributed in all the monitorable objects. Heartbeat messages generated by a large number of monitorable objects may also inadvertently flood the network, while this situation is easier to detect and

avoid if the messages are sent from less sources.

On the other hand, using push style communication between monitors and monitorable objects is more efficient and may reduce the number of messages generated when using hardware multicast facilities (such as IP multicast) if several monitors are listening to the heartbeats.

The type of interaction to use between monitors and clients depends on the application itself, and on the admissible latency of failure information. If the client application has to be notified immediately upon component failure, the push communication paradigm is better. This scheme reduces the number of messages delivered (the number is even minimal, as only state changes are pushed), since the client does not have to poll for changes in the status of monitored objects. In addition, an event-driven application may need to be asynchronously notified about suspicions, which are considered as incoming events.

Conversely, if the client only needs to check the monitorable status at specific points in time, pull communication may be more appropriate, since push communication could generate unnecessary messages. In addition, the push model imposes a server role on the client, and introduces the risk that a notification may *not* arrive at the client, e.g., if there is a network partition between the client and the monitor (this is minor problem with the pull model, because a client can consider an object as suspected if it is not able to communicate with the failure detector that monitors the object).

Notice that both types of interactions can be used at the same time between monitors and clients: the client can check synchronously for the status of a monitored object, while being registered by the monitor for asynchronous notifications of status changes.

The dual approach combines the advantages of both approaches and provides more flexibility by letting monitorable objects use the better suited approach. The trade-off is the slightly increased complexity of the implementation.

Object Monitoring as a Specialization of Event Channels. Another architectural approach to object monitoring consists in reusing the CORBA event service (see Section 1.5.4), which provides the two basic push and pull interaction models. Failure detectors would be specific implementations of event channels, and monitorable objects would be consumers or suppliers depending on the monitoring model. Although some mechanisms of the event service may be directly reused, this approach requires extending the event channel interface to match the failure detection problem (e.g., to support inquiries about the status of monitored object). We did not follow this approach because the event service has been standardized by the OMG and modifying its specification conflicts with our goals.

3.3.3 Design

This section presents the monitoring service design. We describe two different approaches to illustrate how object monitoring can be provided in a CORBA environment. The first one is a simple pull-style service, which was used in early versions of OGS, while the second one is a more comprehensive and flexible service based on the dual model, used in the current version of OGS. Both illustrate interesting aspects of object monitoring, and they can be used interchangeably in OGS.

Similarly to the other OGS components, several concerns have influenced the design of the monitoring service. In addition to the goals common to all services, such as simplicity, orthogonality, or modularity, our design has been guided by several concerns related to object monitoring:

- Reduced network usage (in number of messages).
- Latency of state propagation.
- Encapsulation of failure detector parameters (e.g., timeouts).
- Scalability (to a large number of clients or monitorable objects), and applicability in geographically dispersed environments.
- Adaptability to different network configurations and qualities of service.
- Design genericity, such that different monitoring models are implementable with few design changes.
- Granularity: the monitoring service should monitor *objects* rather than processes or hosts. The programmer can choose to install one or several monitorable objects per host or per process depending on the requirements of the application.

These design goals and principle have been taken into account during the specification of the monitoring service. In particular, scalability has been a major concern when designing the service interfaces, as discussed below.

Approach 1: A Simple Pull-Style Monitoring Service

This section presents the design and architecture of a simple pull-style monitoring service. The interfaces are intentionally minimal to illustrate the underlying concepts in a simple way.

Basic Service Interfaces. When working with distributed objects, one has to define methods that can be invoked remotely. In contrast with a local invocation, a remote invocation can fail because of a communication failure. Therefore, a straightforward way to implement a failure detector using the pull model is to have monitored objects support a specific operation (e.g., `are_you_alive()`) which

is invoked periodically by the failure detector. If the monitored object is alive and there is no communication failure, the invocation succeeds. If the invocation fails, the failure detector suspects the monitored object. This model corresponds to the IDL interfaces of Figure 3.22.

```
1  // IDL
2  enum Status { SUSPECTED, ALIVE, DONT_KNOW };
3
4  interface Monitorable {
5    boolean are_you_alive();
6  };
7
8  interface Monitor {
9    void start_monitoring(in Monitorable mon);
10   void stop_monitoring(in Monitorable mon);
11   Status is_it_alive(in Monitorable mon);
12 };
```

Figure 3.22: IDL Interfaces for the Simple Pull-Style Monitoring Service

To check if an object is alive, the failure detector invokes the `are_you_alive()` operation (line 5) of each monitorable object. If the invocation succeeds and returns `true`, the object is considered as alive; otherwise, it is suspected. The monitored object may also return `false` to simulate a failure (e.g., for debugging purpose). This invocation may be performed:

- On demand (lazy evaluation): the monitorable object is checked on client demand (i.e., when the client asks the monitor for the status of an object). This makes the system less reactive since the client has to wait for the invocation to return before knowing the object's status.
- In background: the failure detector periodically checks if the object is alive, and stores this information in a local table. The failure detector acts as a cache of suspicion information. This information may have a *time-to-live* which tells when to invalidate and re-evaluate the suspicion information.

A client asks the failure detector to start and stop monitoring an object by invoking the `start_monitoring()` and `stop_monitoring()` operations (lines 9–10), and obtains the status of an object by invoking the `is_it_alive()` operation (line 11). A monitored object can have one of three states:

- *SUSPECTED* means that the object is suspected by the failure detector.
- *ALIVE* means that the object is considered as alive by the failure detector.
- *DONT_KNOW* means that the failure detector is not monitoring the object.

A Hierarchical Configuration. The problem of scalability is a major concern for a monitoring service that has do deal with large systems. A traditional approach to failure detection is to augment each entity participating in a distributed protocol with a local failure detector that provides it with suspicion information. However, this architecture raises efficiency and scalability problems with complex distributed applications, in which a large number of participants are involved. In fact, if each participant monitors the others using point-to-point communication, the complexity of the number of messages is $O(n^2)$ for n participants. Wide area communication is especially costly and increases the latency of the whole system. Therefore, it is very important to reduce the amount of data exchanged across distant hosts.

A typical network configuration consists of several Local Area Networks (LANs), with gateways connecting them, and the participating hosts distributed over all these LANs. Inter-LAN communication is more costly (in terms of resources and performance) and generally less reliable than intra-LAN communication. Therefore, it is sometimes preferable to install only two or three failure detectors[5] in each LAN, independent of the number of objects they monitor.

The simple interfaces of our pull-style monitoring service make it easy to configure the monitoring system in a hierarchy, as shown in Figure 3.23. The hierarchy is arranged in a *Directed Acyclic Graph* (DAG). In a LAN, one or several failure detectors can keep track of the state of all local monitorable objects, and transmit status information to remote failure detectors in other LANs, thus reducing the number of costly inter-LAN requests. The hierarchical configuration permits a better adaptation of failure detector parameters (such as timeouts) to the topology of the network or to the distance of monitored objects, and reduces the number of messages exchanged in the system between distant hosts. A failure detector located in a LAN can adapt to the network characteristics and provide a specific quality of service. The reduction of network traffic, especially when a lot of monitorable objects and clients are involved, is the main reason for the good scalability of this hierarchical approach. On the other side, the latency of the state propagation may increase with the indirections caused by the hierarchical configuration.

In the hierarchical configuration of Figure 3.23, two groups of objects ($M3$ and $M3'$) are both being monitored by two distinct monitors ($FD3$ and $FD3'$) in a LAN ($LAN3$). Two clients ($C1$ and $C1'$) are located in another LAN ($LAN1$), and monitor the former objects indirectly through a local monitor ($FD1$). There are two distinct paths between $FD1$ and the monitorable objects, making the failure of $FD3$ or $FD3'$ transparent to the clients. However, there is no redundancy in $LAN1$, and the failure of $FD1$ would prevent clients from getting liveness information about $M3$ and $M3'$. This configuration example reduces inter-LAN communication, when compared to a traditional approach with one local monitor per client, and messages exchanged between each monitor and monitorable object.

Since a link may break anywhere in the hierarchy, a set of simple rules for hierarchical invocations helps determining if a particular object is suspected or not:

[5] This redundancy is necessary for a fault tolerant system.

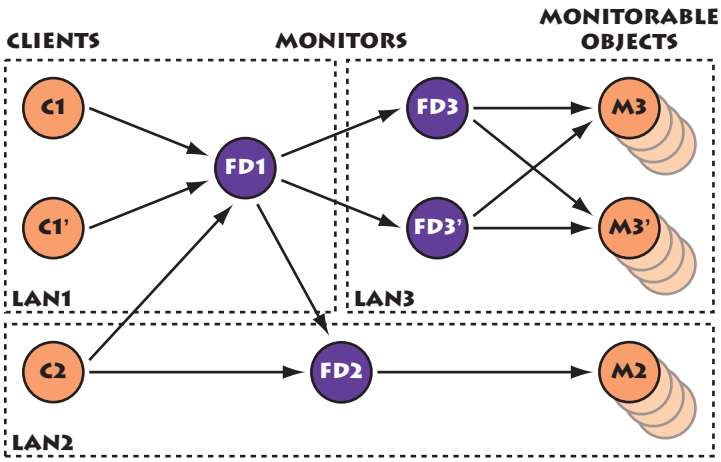

Figure 3.23: A Typical Hierarchical Configuration

- If a monitor says that a monitorable object is alive, this object was actually alive some time before.
- If an invocation to a monitor fails when asking for the status of a monitorable object, the invoker must assume that the object is suspected by the monitor.
- If there is more than one path leading to a monitorable object, and this object is not suspected by the monitors of at least one path, it must be considered as alive.

Providing a hierarchical configuration requires adding management operations for linking and unlinking monitors, finding invocation paths between clients and monitorable objects, etc. Describing these management functions is not relevant in the context of this dissertation.

Monitorable as a Specialization of Monitor. With this simple pull model and its hierarchical configuration, it is possible to provide a clean and orthogonal design in which monitorable objects are a specialization of monitor objects. In fact, a monitorable object can be viewed as *a monitor object that only monitors itself, and that never suspects itself.* A call to `mon->are_you_alive()` would be replaced by `mon->is_it_alive(mon)`. This change has no impact on the previously presented architecture, assuming that the rules for hierarchical invocations are respected. The new interfaces would be simpler, since there would be no monitorable object. We will not detail this approach further, but it has a big advantage of providing a simple, clean, and orthogonal view of object monitoring.

Adding Asynchronous Suspicion Notifications. Although most applications need to invoke the failure detector synchronously at specific points during protocol execution, it may be sometimes useful to receive asynchronous notifications when

the state of an object changes. Figure 3.24 presents the interfaces of the simple pull-style monitoring service, extended with asynchronous notifications support. An extra parameter has been added to the `start_monitoring()` operation (lines 14–15), that allows us to register an object with the `Notifiable` interface (lines 8–11). The failure detector invokes the `notify_suspicion()` operation (lines 9–10) of each registered notifiable object when the status of a monitored object changes. The client may still pass a null reference as notifiable object if it is not interested in asynchronous notifications.

```
1  // IDL
2  enum Status { SUSPECTED, ALIVE, DONT_KNOW };
3
4  interface Monitorable {
5    boolean are_you_alive();
6  };
7
8  interface Notifiable {
9    void notify_suspicion(in Monitorable mon,
10                         in boolean suspected);
11 };
12
13 interface Monitor {
14   void start_monitoring(in Monitorable mon,
15                         in Notifiable not);
16   void stop_monitoring(in Monitorable mon,
17                        in Notifiable not);
18   Status is_it_alive(in Monitorable mon);
19 };
```

Figure 3.24: IDL Interfaces for Asynchronous Suspicion Notifications

Approach 2: An Extended Multi-Style Monitoring Service

This section presents the interfaces of the OGS monitoring service that supports the dual model, i.e., a combination of the push-style and pull-style monitoring models (Figure 3.25). It reuses some of the interfaces presented in Section 3.3.3.

Push-style monitoring is performed via a pair of one-way operations, `i_am_alive()` and `are_you_alive()` (lines 24 and 31), instead of a single two-way operation. While this has little impact on the implementation, it allows us to combine the push and pull model in a clean interface hierarchy. Notice that CORBA one-way invocations provide only best effort semantics. A better solution would be to use OMG's future messaging service [BEI+98], which will provide various qualities of service such as asynchronous reliable communication.

The client interfaces to the monitoring service, defined by the `Monitorable`, `Notifiable` and `Monitor` base interfaces (lines 4–20), abstract the flow model used for object monitoring. The client does *not* need to know which model is supported and implemented by the monitoring service. The push and pull models are defined by subtyping client interfaces and adding operations for monitoring objects (lines

```
1  // IDL
2  module mMonitoring {
3    // Client interfaces for all flow models
4    enum Status { SUSPECTED, ALIVE, DONT_KNOW };
5
6    interface Monitorable {
7    };
8
9    interface Notifiable {
10     void notify_suspicion(in Monitorable mon,
11                           in boolean suspected);
12   };
13
14   interface Monitor {
15     void start_monitoring(in Monitorable mon,
16                           in Notifiable not);
17     void stop_monitoring(in Monitorable mon,
18                          in Notifiable not);
19     Status is_it_alive(in Monitorable mon);
20   };
21
22   // Interfaces for all flow models
23   interface HeartbeatMonitor : Monitor {
24     oneway void i_am_alive(in Monitorable mon);
25   };
26
27   // Pull model
28   interface PullMonitor : HeartbeatMonitor {};
29
30   interface PullMonitorable : Monitorable {
31     oneway void are_you_alive();
32   };
33
34   // Push model
35   interface PushMonitor : HeartbeatMonitor {};
36
37   interface PushMonitorable : Monitorable {
38     void send_heartbeats(in PushMonitor mon,
39                          in long frequency);
40   };
41
42   // Dual model
43   interface DualMonitor : PullMonitor,
44                           PushMonitor {};
45
46   interface DualMonitorable : PullMonitorable,
47                               PushMonitorable {};
48 };
```

Figure 3.25: IDL Interfaces for the Extended Multi-style Monitoring Service

23–40). The dual model is defined in a clean way by simply inheriting from the push and pull models (lines 43–47).

The class diagram of Figure 3.26 illustrates the interfaces and operations of the complete object monitoring service specification in a class diagram. Client applications that use the service for monitoring remote objects have a limited view of the service, restricted to the three topmost interfaces. Application-specific servers that implement monitorable objects may choose to inherit from any of the `PullMonitorable`, `PushMonitorable`, or `DualMonitorable` interfaces.

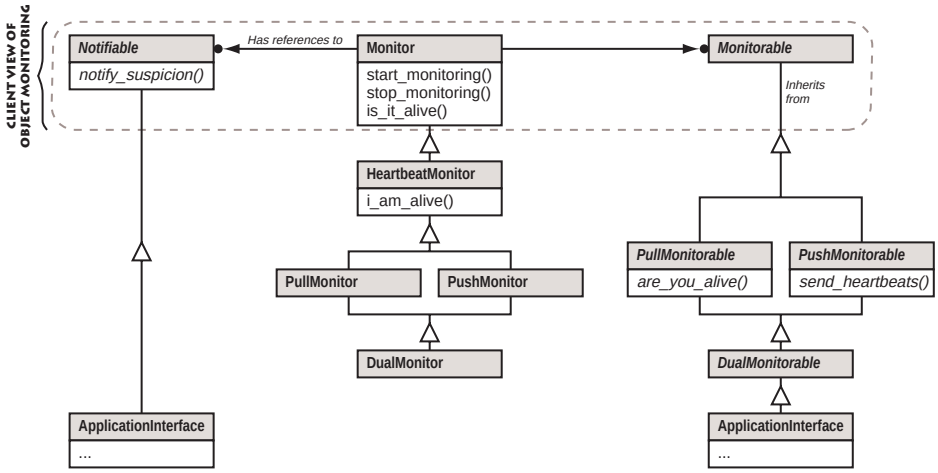

Figure 3.26: Class Diagram of the Object Monitoring Service Interfaces

3.3.4 Applicability of the Service

The monitoring service is useful for numerous application domains. For instance, supervision and control systems must continuously monitor the status of the components in the system and react to the failure of one of them. Garbage collecting of distributed resources requires monitoring facilities as well (e.g., a distributed lock must be released if the entity that holds it has failed).

Similarly to the consensus service, the monitoring service may also be used as a building block for the implementation of distributed protocols. In fact, several distributed algorithms are based on the asynchronous system model augmented with an unreliable failure detector; the monitoring service provides failures detection mechanisms that may be directly reused in this context.

⋆
⋆ ⋆

3.4 The Messaging Service

An environment like OGS that provides support for reliable distributed computing requires reliable communication channels. Reliable communication between two objects may be defined as follows:

> *If both the source and the target objects do not fail, then the message will be eventually delivered to the target object.*

This definition requires potential link failures between the source and target objects to be eventually repaired. In addition to reliability, the protocols used for reliable computing generally require the communication primitives to be asynchronous, i.e., the client is *not* blocked while the invocation is processed.

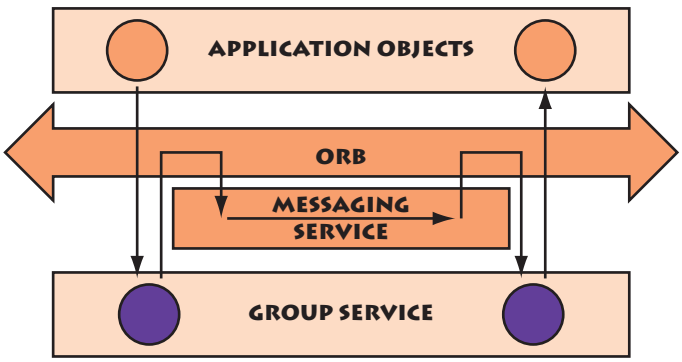

Figure 3.27: The Messaging Service

These two requirements cause several group communication systems to use proprietary communication mechanisms based on datagram protocols, instead of connection-oriented protocols such as TCP/IP. Standard CORBA invocation mechanisms do not provide the required degree of reliability and asynchrony. For this reason, we have encapsulated the problem of asynchronous reliable communication in one component: the *Object Messaging Service* (Figure 3.27). This service sits close to the ORB, and is used by OGS for communication between remote components. This section presents the problems related to asynchronous reliable communication and the solutions adopted in the OGS environment.

3.4.1 Standard CORBA Invocation Mechanisms

CORBA remote object invocations are based on RPC-like mechanisms and are, by default, synchronous, i.e., the client is blocked until the invocation returns. This behavior is convenient for many applications, but is unsuitable when, for instance, a client has to perform several invocations to different servers, and the servers or the links are slow; in this situation, the client should issue all requests at once and let them execute in parallel while waiting for replies.

In addition to synchronous invocations, the CORBA specification allows the declaration of *one-way* operations: the operation does not return any value and the client is not interested in waiting for the completion of the invocation. The specification also defines primitives for sending *deferred synchronous* invocations: the client thread continues processing, subsequently polling to see if results are available. Currently, this model is only available when using the *Dynamic Invocation Interface* (DII).

Nevertheless, one-way invocations are *not* really asynchronous.[6] The term *one-way* specifies that the client will not wait for the completion of the operation on the server side (which can last a long time depending on the application), but it does not require the ORB to use non-blocking communication channels; thus, a one-way call can block "forever" *at the transport level*, which can block in turn the application. For instance, an *Internet Inter-ORB Protocol* (IIOP) one-way invocation can block the entire process on a one-way call just because a TCP/IP buffer fills up. According to the CORBA specification [OMG98a], *given TCP/IP's flow control mechanism, it is possible to create deadlock situations between clients and servers [...] ORBs are free to adopt any desired implementation strategy, but should provide robust behavior.* Although an ORB implementation could detect a possible deadlock when performing an IIOP call, this behavior is not guaranteed by the CORBA standard.

Concerning reliability, the semantics of standard CORBA invocation mechanisms have been kept (intentionally) vague in the specification. Two styles of execution semantics are defined by the object model [OMG98a]:

- *Exactly once* or *at-most-once:* if a two-way operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once.
- *Best-effort:* a best-effort operation is a request-only operation, i.e., it cannot return any result and the requester never synchronizes with the completion, if any, of the request.

The second execution style does not provide any guarantee. A *compliant* ORB can just send the invocation message once, or even discard it. If the message is sent more than once, this can result in multiple invocations on the server and lead to consistency problems with non-idempotent operations. Furthermore, the CORBA specification is completely silent on what happens in case of a link failure, if the network is congested, or if the server is extremely busy. Since this behavior is not specified, there is no way to handle it portably.

3.4.2 Overcoming Limitations of Standard CORBA Invocations

In order to overcome the limitations of the standard CORBA invocation model, we have defined an object messaging service that provides basic mechanisms for

[6] In the CORBA terminology, asynchronous means that the client ORB does not synchronously invoke the target to obtain a reply. Some other agent is required to separate the client ORB from the target.

managing *asynchronous* point-to-point messages. Its main purpose is to (1) allow clients to invoke servers without blocking the client execution thread, and (2) allow clients to specify the required quality of service for sending a message. Qualities of service include *unreliable*, *reliable*, and *FIFO* communication. The messaging service isolates the *semantic requirements* of OGS concerning remote communication. These requirements will be fulfilled by the OMG's future messaging service (see Chapter 5.6), and we intend to use this service in OGS as soon as implementations become available.

3.4.3 Support for Reliable Multicast

In addition to reliable point-to-point communication, group communication systems require reliable multicast facilities. Reliable multicast may be defined as follows:

> *If the source object does not fail or if one correct target object delivers the message, then the message is eventually delivered by all correct target objects.*

Multicast primitives have been defined in the messaging service to allow provision of one-to-many communication primitives using highly-efficient protocols based on hardware multicast facilities. Using multiple point-to-point invocations is slower than with hardware multicast when there are a large number of destinations and does not scale well to hundreds of destinations. The multicast facilities of the messaging service abstract the underlying communication mechanisms used for implementing reliable multicast. These primitives can easily be implemented using simple protocols based on point-to-point messages or using hardware multicast facilities,[7] without impact on the other OGS components.

Qualities of service provided for multicast include *unreliable* and *reliable* multicast, with *FIFO* channels between the source and the target objects, but without ordering guarantee for concurrent multicasts to the same target objects. The messaging service does not know about object groups; group multicast is provided in the group service with various ordering guarantees (see Section 3.1).[8]

⋆
⋆ ⋆

[7] This choice is implementation dependent. Notice that using hardware multicast does not comply with the IIOP standard, and prevents applications from interoperating with ORBs that are not adapted to use the hardware multicast.

[8] Note that OMG's future messaging service does not define multicast communication primitives.

Summary

The OGS architecture defines several distinct services that are used to provide group communication facilities in a CORBA environment. These services are specified in terms of IDL interfaces, and are independent of one another.

The group service is the core of the OGS environment. It implements essentially two functionalities: *group membership* manages the life cycle of object groups, and *group multicast* provides support for sending requests to all members of a group with various guarantees. The group service offers several levels of transparency and reliability. Unlike traditional group communication toolkits, OGS lets the server specify the semantics associated with each client request.

The consensus service allows a set of CORBA objects to solve the distributed consensus problem. The consensus problem is a central abstraction that can be used to solve various distributed problems, and in particular to implement group communication protocols. The consensus service defines IDL interfaces for running consensus protocols (i.e., to instantiate a consensus, to propose an initial value, and to receive a decision), independently of the algorithm used to solve consensus.

The monitoring service defines interfaces for detecting remote component failures. Several models can be used for object monitoring. The most common ones are the push and the pull models, which can be combined in a single, generic model: the dual model. In addition, the objects of the monitoring service can be organized in a runtime hierarchy that reduces the number of messages exchanged in the system by the failure detection mechanisms.

Finally, the messaging service is a component that isolates the requirements of OGS concerning remote communication. It will be replaced by OMG's future object messaging service.

Chapter 4

The Object Group Service: Implementation Issues

If a series of events can go wrong, it will do so
in the worst possible sequence.

The extended Murphy's law

An implementation of OGS has been developed in the context of this thesis. It was first developed in C++ using Orbix 2.xMT [ION97], and then ported to VisiBroker for C++ 3.x [Vis98a]. A Java prototype has also been implemented with VisiBroker for Java 3.x [Vis98b].

This chapter presents the current implementation of the services composing the OGS environment. We present the system model, and the general principle adopted by OGS for distributed protocol support. We give an overview of the implementation dependencies between the OGS components, and describe the different algorithms used by these components. We present implementation issues, such as transparency support and group naming, and we discuss some of our experiences when implementing and porting OGS. These experiences illustrate several shortcomings of the current CORBA specification. Finally, we present performance measurements of the OGS implementation.

4.1 System Model

When dealing with distributed systems, it is useful to distinguish between *asynchronous* and *synchronous* computing systems [Sch93b]. In the asynchronous model, no assumption is made about the speed of the communication system and the processes in the network. There is no bound on the time that it may take for a message to reach its destination process.

On the other hand, the synchronous model assumes bounds on communication delays and on the relative speed of processes. In practice, postulating that a system is synchronous introduces strong constraints on how processes and communication channels are implemented. Fully synchronous systems have severe limitations regarding the nature, the location, and the number of the machines that compose the system. Since the asynchronous model makes fewer assumptions about the underlying system, a protocol designed for use in an asynchronous system can be used in a synchronous one. Therefore, only the more general asynchronous model has been be considered in the context of this work.

Fischer, Lynch, and Paterson demonstrated that the consensus problem cannot be solved in fully asynchronous systems if only one participant of the consensus protocol fails [FLP85]. Later, Chandra and Toueg have proposed a protocol that solves the consensus in an asynchronous system augmented with an unreliable failure detection mechanism [CT96]. In OGS, we use the monitoring service (see Section 3.3) together with the protocol of Chandra and Toueg to solve distributed agreement problems.

4.2 Distributed Protocol Support in OGS

The entities that participate in a distributed algorithm must communicate with each other, as part of a *distributed protocol*. The term "protocol" is used to refer to a set of precisely-defined rules and conventions used for communication between distributed components [CD88]. OGS components execute distributed protocols by issuing remote invocations to one another.

CORBA does not differentiate between local and remote invocations issued to IDL-specified operations. As a matter of fact, stubs and skeletons hide the distribution from the application programmer, and allow both local and remote implementations. Although this programming model bears many advantages, the developer of fault tolerant services has to deal with link failures and independent component crashes; these events have effects on the behavior of the system depending on whether the components are remote or local.

In the OGS implementation, we make assumptions about the locality of some service-specific objects. For instance, the consensus manager, the monitorable object, and the group administrator which are associated with each group member are located in the same process. Local invocations are reliable, and we assume that independent failures of co-located objects do not occur.[1]

To deal with the problem of remote invocations and failures, we have introduced an *abstract* messaging service that defines better quality of service than standard CORBA remote invocations (see Section 3.4). This service appears at the architectural level, but it is not defined in terms of IDL interfaces and is not implemented in the OGS prototype. We plan to use OMG's messaging service for this purpose as soon as it becomes available. In our current implementation, we assume that the ORB communication primitives use reliable channels, and that only crash failures

[1] Deadlocks in a multi-threaded application may have a behavior similar to independent failures.

may prevent an object from receiving a message. In other words, we expect links to be reliable. Since the IIOP protocol used by the ORB implementation is based on TCP/IP, this assumption is reasonable in LANs.

Reliable Communication in Spite of Link Failures

As just mentioned, OGS does *not* implement reliable mechanisms for remote invocations; instead, it uses the primitives provided by the ORB as a temporary substitute for OMG's future messaging service. However, it is possible to implement reliable non-blocking communication using existing CORBA mechanisms, in spite of link failures.

A simple solution to reliable non-blocking communication is to use multi-threading on the client side, with a separate thread for each invocation. Each time a remote invocation has to be issued, the client forks a new thread, and uses a standard CORBA synchronous invocation. If the invocation fails, the client re-issues it until the invocation succeeds or the server is considered to be failed. Depending on the exception returned to the client upon remote invocation failure, there may be no possibilities to tell whether the server actually received the invocation or not; in such a situation, at-most-once semantics can be guaranteed by adding a sequence number to each invocation.

Although this algorithm implements reliable non-blocking communication, the use of threads considerably increases the application's complexity and the probability of programming errors, by adding resource and synchronization problems. In addition, the price to pay in terms of lost efficiency is tremendous. Therefore, we avoided this solution in the context of OGS.

Public Interfaces vs. Protocol Interfaces

The OGS components exhibit well defined interfaces (see Chapter 3) through which applications may request services. Since OGS components have to run distributed protocols without the knowledge of the application, they must have some private way to communicate with each other. For this purpose, we introduce the notion of *protocol interfaces* which are private, implementation-specific interfaces known only by the service, that are not accessible to the application. OGS components use these interfaces to exchange the messages required for executing distributed protocols, while public interfaces offer application-specific services.

Figure 4.1 illustrates public and protocol interfaces in the context of the group service. The `GroupAccessor` and `GroupAdministrator` interfaces are public and exported to the application, while the `GroupAccessorProtocol` and `GroupAdministratorProtocol` interfaces are private and used by the group service to execute its distributed protocols. The same model has been adopted for the implementation of the other services.

In the current implementation of the OGS components, the protocol interfaces are subclasses of the public component interfaces. Alternatively, the use of multiple

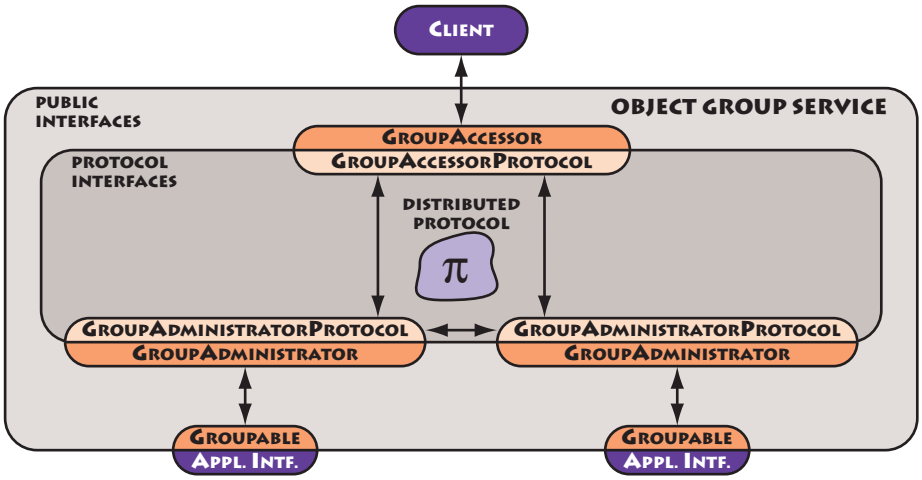

Figure 4.1: Public and Protocol Interfaces of the Group Service

inheritance from public and protocol interfaces would have been possible.

Component Dependencies

Although OGS is not organized in a layered architecture, there are dependencies between its components that are very much like inter-layer relationships. These dependencies are *implementation-specific*, and do not appear in the IDL interfaces of the services. The relationships between the different components are illustrated in Figure 4.2:

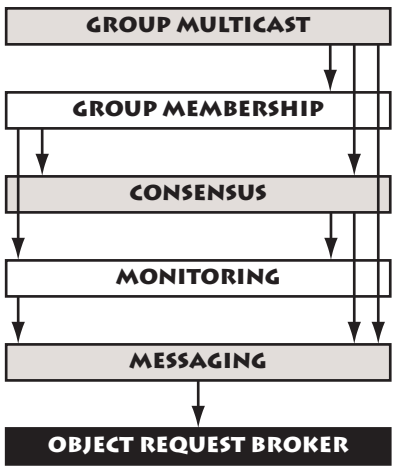

Figure 4.2: Dependencies between OGS Components

- The messaging service defines OGS' requirements concerning reliable point-to-point and multicast communication.

- The monitoring service uses reliable communication for detecting object failures.
- The consensus service uses failure detection and reliable communication mechanisms to solve the distributed consensus problem in an asynchronous environment augmented with failure detectors.
- Group membership uses failure detection mechanisms to monitor group members and a consensus protocol to agree on new views.
- Group multicast uses reliable communication, consensus, and group membership for atomic message delivery to all members of a group.

Protocol Dependencies

Protocol interfaces reflect the dependencies between the different OGS components, by supporting operations of several components. As an example, the `GroupAdministratorProtocol` interface plays a central role in OGS. The objects implementing that interface are responsible for all the group communication support on the server side. They combine the functionalities of all OGS components by inheriting from interfaces defined by these components (Figure 4.3): they execute the group multicast and group membership protocols (group accessor and group administrator), they monitor each other in order to detect member failures (monitorable and notifiable), and they participate in the consensus protocols (consensus participant). The resulting protocol interface is used internally in the OGS implementation. The same implementation principle applies to the other protocol classes of OGS.

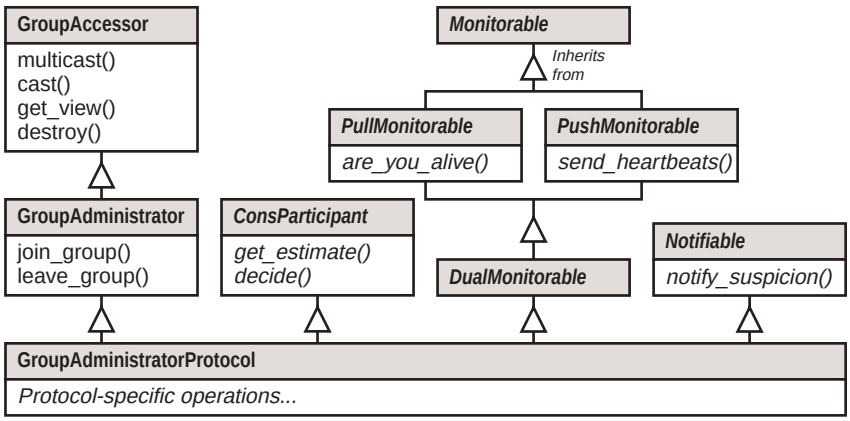

Figure 4.3: Class Diagram of a Protocol Interface

4.3 Group Communication Algorithms

This section presents the algorithms and protocols used in OGS to implement group communication. The algorithms are expressed in terms of interactions between sets

of *processes*, although OGS considers interactions between sets of *objects*. No formal proof is given here: most of the proofs are obvious, or may be found in the referenced papers.

In the algorithms, we assume that for each message m, there is a function $sender(m)$ that returns the sender of the message. In addition, unlike the formal specification of the algorithms, the implementation uses message identifiers extensively (instead of messages) to reduce network utilization to the minimum.

4.3.1 Failure Detection

The protocol used in OGS to implement failure detection is based on adaptive timeout mechanisms. Although OGS provides two versions of the monitoring service — one based on the pull model and the other on the dual model, which combines the push and pull models (see Section 3.3.2) — the current implementation uses only pull-style communication. Failure detectors periodically check monitorable objects by sending them requests, and suspect them if they do not receive a reply within specific time bounds. Although this protocol does not ensure eventually weak accuracy in a truly asynchronous system,[2] it is adequate *in practice*, when the timeouts are appropriately defined.

4.3.2 Consensus

The implementation of the OGS consensus service is based on the algorithm of Chandra and Toueg [CT96]. This algorithm solves the consensus problem in an asynchronous system augmented by an unreliable failure detection mechanism (called $\Diamond\mathcal{S}$) under the condition that a majority of the participating processes do not fail.

The algorithm, described in [CT96], is based on the rotating coordinator paradigm, and proceeds in asynchronous rounds. In every round, a different process plays the role of the coordinator. A round consists of four sequential phases:

I. Every process sends its current estimate to the current coordinator. In the first round, the estimate is the initial value of the process.[3]

II. The coordinator of the current round waits for a majority of estimates and selects one with the highest timestamp (the timestamp represents the last round in which a process changed its estimate). That estimate is sent to all processes.

III. Every process waits for the new estimate proposed by the current coordinator. Once a process receives the new estimate, it sends back an acknowledgement message (ACK) to the coordinator. However, if the process does not receive the estimate and the coordinator is suspected by the failure detector, a negative acknowledgement (NACK) is returned to the coordinator.

[2] *No* protocol can ensure eventually weak accuracy in a truly asynchronous system.

[3] A simple optimization of the consensus algorithm consists in skipping the first phase of the first round, and having the first coordinator directly propose its estimate to all processes.

IV. The coordinator waits for the acknowledgements from a majority of processes. If none of those acknowledgements is a NACK, the coordinator decides its current estimate and reliably broadcasts the decision to all processes.

A good run (i.e., without crash or failure suspicion) of the algorithm with three participating processes is depicted in Figure 4.4.[4]

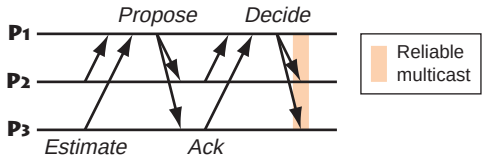

Figure 4.4: Good Run of the Consensus Algorithm

The implementation of the OGS consensus service is fully event-driven. It is implemented using a state-machine approach, and reacts to two types of events: message receptions and suspicion notifications. The different states correspond to the different phases of the algorithm. The implementation allows multiple parallel executions of the consensus. A participant finishes the algorithm once it has decided on some value. Parallel or consecutive consensus instances may be run with completely disjoint sets of participants.

Extended Consensus

In order to fulfill the requirements of the view membership algorithm presented in Section 4.3.4, we have extended Chandra and Toueg's consensus algorithm so that it can *optionally* decide on a non-empty set composed of a majority of initial estimates, instead of a single estimate. To reflect this new property, the original algorithm has been modified as follows: in the second phase of the consensus algorithm, the coordinator proposes a *list* of all the estimates received from the other participants, if all these estimates have a timestamp of zero; otherwise, it selects one estimate with the highest timestamp, as in the original algorithm. The validity rule of the consensus problem (see Section 3.2.1) must be changed as follows:

- *List Validity:* If a process decides V, then V is a non-empty list containing a majority of initial values, each of which has been proposed by some process.

4.3.3 Reliable Group Multicast

The algorithm used by OGS for reliable multicast is based on the reliable broadcast algorithm proposed by Chandra and Toueg in [CT96]. This algorithm uses message

[4]Notice that the coordinator also sends messages to itself in each phase, although they are not depicted in the figure.

re-diffusion to ensure that every correct destination process actually receives the message.

The idea of this algorithm is the following: when a process receives a reliable multicast message for the first time, it relays the message to all other destination processes, and then delivers it to the application. Although not very efficient, this mechanism ensures that all correct processes deliver every message. The algorithm is defined in terms of two primitives: *R-multicast*, invoked by the issuer of the multicast, and *R-deliver*, invoked asynchronously on the receivers. It is formally described in Figure 4.5 (in the context of group communication, *dsts* designates a group instead of a set of processes).

```
1: {Protocol of the client}
2: procedure R-multicast (m, dsts)
3:    send (m, dsts) to dsts                    {dsts is a set of processes (or a group)}

4: {Protocol of the server (code of process p)}
5: when receive (m, dsts) for the first time
6:    if sender(m) ≠ p then
7:       send (m, dsts) to dsts\{p, sender(m)}
8:    R-deliver (m)
```

Figure 4.5: Reliable Multicast Algorithm [CT96]

A variation of this algorithm, more efficient in terms of resource consumption, consists in re-diffusing a message *only* if the message has not been stabilized after some time: stabilization information can be piggybacked on subsequent messages to the same group. This optimization is not implemented in the current prototype of OGS.

4.3.4 Group Membership and Total Order Multicast

OGS uses the consensus service for implementing group multicast and group membership. The role of the consensus is to agree on the respective ordering of the events received by group members. These events are *messages* and *view changes*. Hence, each consensus instance decides on:

- An ordered set of messages to deliver.
- A set of suspected members to remove from the current view (i.e., the composition of the next view). This set can be empty, in which case there is no view change.

The consensus algorithm ensures that all correct participants eventually decide on the same value, and thus that every group member delivers the same set of messages and view changes in the same order. Totally ordered messages are reliably multicast

```
1: {Protocol of the client}
2: procedure TO-multicast (m, dsts)                    {Issue total order multicast}
3:    R-multicast (m, dsts) to dsts                    {dsts is a set of processes (or a group)}

4: {Protocol of the server (code of process p)}
5: Initialization:
6:    delivered_p ← ∅                                  {Messages already TO-delivered}
7:    unordered_p ← ∅                                  {Messages not yet ordered}
8:    suspected_p ← ∅                                  {Suspected members from the current view}
9:    cid_p ← 0                                        {Consensus identifier}

10: when R-deliver (m)                                 {Receive total order message}
11:    if m ∉ delivered_p then
12:       unordered_p ← unordered_p ∪ {m}

13: when suspect (q)                                   {Suspect a group member}
14:    suspected_p ← suspected_p ∪ {q}

15: when unsuspect (q)                                 {Trust a group member again}
16:    suspected_p ← suspected_p \ {q}

17: when unordered_p ≠ ∅ or suspected_p ≠ ∅            {Order messages and views}
18:    k ← cid_p
19:    propose (k, {unordered_p, suspected_p})         {Launch consensus}
20:    wait until decide (k, {unordered_k, suspected_k})
21:    atomically TO-deliver all messages in unordered_k in some deterministic order
22:    delivered_p ← delivered_p ∪ unordered_k
23:    if suspected_k ≠ ∅ then
24:       install new view without the processes from suspected_k
25:    unordered_p ← unordered_p \ unordered_k
26:    suspected_p ← suspected_p \ suspected_k
27:    cid_p ← cid_p + 1
```

Figure 4.6: Total Order and View Membership Algorithm

in the group before being ordered by the consensus algorithm.[5]

The algorithm for total order and view membership is presented in Figure 4.6. Similarily to the total order algorithm of [CT96], each participant maintains a set of unordered messages ($unordered_p$) updated each time a message is received. The list of the suspected members from the current view ($suspected_p$) is updated each time the failure detector notifies the participant about a suspicion or an "unsuspicion" (i.e., a suspected participant is trusted again). The cid_p variable is used as consensus identifier to synchronize the consensus instances run by all participants. A consensus is launched when there are messages to order, or there are members to remove from the current view. Each participant delivers the set of messages contained in the decision in a deterministic order that was agreed *a priori* by all participants.

[5]The reliable multicasting of the message is not necessary if the full message (instead of just the message identifier) is included in the participants' estimates. In addition, OGS can be configured to reliably multicast the message only if it is not included in the decision of the following consensus, or if it has not been delivered within some delay. This optimization has been implemented for the performance measurements of Section 4.6.

If the set of suspected members contained in the decision is not empty, all members install a new view that does not include the suspected members. The process of joining or leaving an existing group is slightly different than that of removing a failed member. It is implemented through a totally ordered request issued by the member joining or leaving the group. This request is processed by every group member, which updates its view accordingly. Joining members receive service-specific information as part of the state transfer protocol (e.g., the composition of the group, the identifier of the next consensus to execute, etc.).

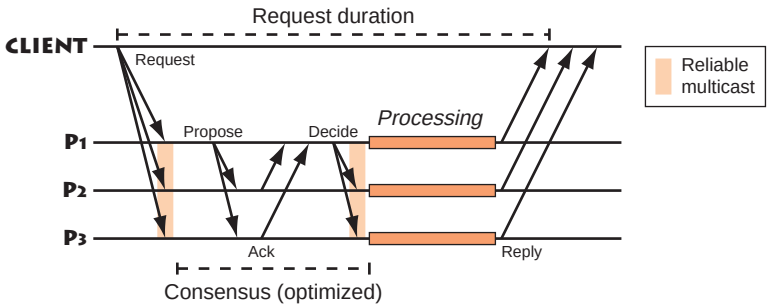

Figure 4.7: Good Run of the Consensus Based Total Order Algorithm

Figure 4.7 ilustrates a good run of the total order algorithm with one client and three group members, using an optimized version of the original consensus algorithm of Chandra and Toueg (the first phase of the first round has been skipped, as described in Section 4.3.2). The idea of using a consensus algorithm for group communication was used in the Phoenix group communication toolkit [Mal96], and has been generalized in [GS96].

Sequenciality of Consensus Executions

Although our implementation of the consensus service allows us to run several consensus instances in parallel, the OGS group service serializes consensus executions. This is required by the total order and view membership algorithm of Figure 4.6. This restriction does not slow down the system since (1) a consensus may decide on the ordering of several messages at once, and (2) it does not prevent non-totally ordered messages from being delivered.

Launching the Consensus

In the OGS implementation of the consensus algorithm, a consensus must be launched explicitly by all participants (see Section 3.2.2 for a discussion about consensus instantiation). It is initiated in two situations:

1. If there are messages to order.
2. If one or several group members are suspected.

Since all correct members of a group *must* start each consensus, we have to ensure that they all receive an event leading to one of the above situations. This is obviously the case with a totally ordered message, because this message is reliably multicast in the group. Since every correct member eventually receives the message (guaranteed by the reliable multicast algorithm), every correct member eventually starts a consensus.

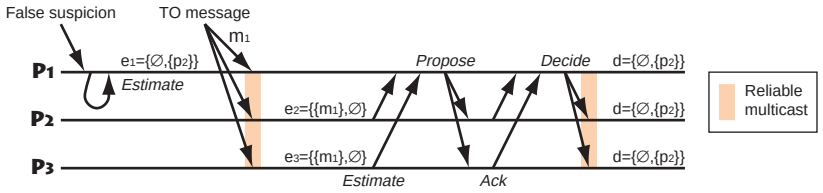

Figure 4.8: Problem with the Consensus Algorithm upon False Suspicion

In the second situation, a false suspicion may lead a member to start a consensus while the other members do not. In Figure 4.8, p_1 incorrectly suspects p_2 and starts a consensus with an initial estimate containing its suspicion information and no message to deliver. It then waits for the other members to start the consensus. The consensus actually starts when the other members have messages to order or suspicions to handle. This delay is not a problem in itself, since $p1$ continues to receive and deliver *unordered* messages. In the figure, an incoming message m_1 causes p_2 and p_3 to start a consensus with an initial estimate containing m_1 and no suspicion information. The consensus is run and the decision value happens to be the initial proposition of p_1, i.e., to remove p_2 from the new view and to deliver no message. This is obviously a bad decision since the removed process has not failed and was suspected by *only* one group member. In addition, totally ordered messages are not delivered to the application, even though all group members have received them.

This unwanted behavior results from the fact that the consensus decides on one value from any participant.[6] To solve this problem, we propose a slightly extended version of the consensus algorithm of Chandra and Toueg, which decides on a *majority of propositions* (see Section 4.3.2). Each participant can then interpret this decision value in a deterministic way:

- Deliver in a deterministic order *all* the messages that are included in the propositions.
- Remove from the group all members that are included in a *majority* of the propositions.[7]

This extension allows the delivery of more messages, and prevents taking wrong actions in case of false suspicion. OGS lets the application developer choose between

[6] Actually, if there is no failure, the decision is the proposition of the first coordinator.

[7] A more conservative approach would be to remove the members that are included in *all* the propositions, and are thus suspected by a majority of group members.

both algorithms.

4.3.5 Optimistic Active Replication

Ordering messages using a consensus algorithm is efficient if several clients are issuing requests concurrently. In this case indeed, one consensus execution can order several messages. On the other hand, if a single client issues several requests sequentially, the cost of the consensus may be considered too high.

To reduce this cost, we propose, for active replication, a new total order algorithm based on a sequencer. We call this algorithm *Optimistic Active Replication* algorithm since it includes the processing of the request by all servers (active replication), and it is optimized for the case when no failure occurs. This algorithm makes the assumption that a majority of processes are correct.

Informally, the algorithm works as follows: a message from a client is first reliably multicast to all participants. Upon message reception, the sequencer — which may be chosen arbitrarily as the first member of the group view — assigns a sequence number to the message and sends this number to all participants. Each participant waits for an ordering information or a suspicion of the sequencer. When a participant receives a sequence number, it delivers the message accordingly, processes the request, and sends the reply with a positive timestamp to the client. If the sequencer is suspected, message ordering is decided using an agreement protocol, and the participants send a reply with a negative timestamp. *The client waits for the replies from a majority of participants that have the same positive timestamp, or for any reply with a negative timestamp.*

The agreement protocol run upon suspicion from the sequencer is the modified consensus algorithm presented in Section 4.3.2. The participants propose the ordering information that they received from the sequencer. The consensus decides on a majority of estimates from all participants. At that point, two situations may arise:

1. *The client has received a majority of replies with the same positive timestamp.* This means that a majority of participants have processed the request prior to starting the consensus, and have thus received the ordering information from the sequencer. In this case, the decision of the consensus contains the ordering information for the message (because the decision contains a majority of estimates), and all participants adopt it.

2. *The client has not received a majority of replies with the same positive timestamp.* If the decision of the consensus contains an ordering information for the message, all participants adopt it; otherwise, they wait for a new sequencer to order the message. The client discards the replies.

After the decision of the consensus algorithm, another sequencer is chosen — which may be the next member in the group view. A good run of this algorithm is shown in Figure 4.9.

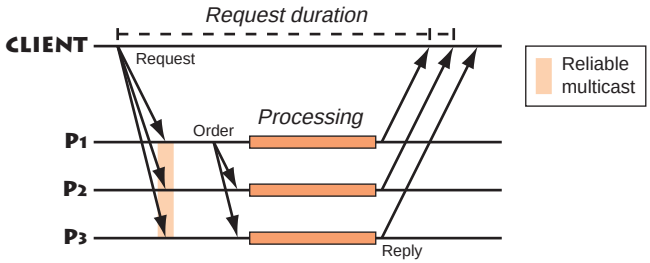

Figure 4.9: Good Run of the Optimistic Active Replication Algorithm

The tradeoff of this algorithm is that, in the worst case,[8] a minority of participants containing the first sequencer may have delivered more messages, or some messages in a wrong order. These particiants have to resynchronize with the primary partition in order to guarantee the correctness of the system. But this situation is extremely unlikely, and this minority will be removed from the view by the membership protocol anyway. Furthermore, the client will discard the replies from this faulty minority of participants.

In this algorithm, the message is not stabilized by the group, but by the client. To stabilize the message in the group, a standard sequencer-based algorithm would require the sequencer to wait for a majority of acknowledgements before sending the reply to the client.

The full algorithm is presented in Figure 4.10. It describes the actions of the client (lines 1–4), of the sequencer (lines 11–15), and of non-sequencer processes (lines 16–36). The initial multicasting of the message (line 3) is not necessary if the sequencer sends the initial message together with its ordering information (instead of just a message identifier). The participants must just ensure that the sequencer has received the message.[9] The performance measurements of Section 4.6 have been performed with this optimization.

4.3.6 Primary-Backup Replication

OGS implements the semi-passive replication technique defined in [DSS98a], which is close to primary-backup replication from the server's point of view. The implementation is based on the *Deferred Initial Values consensus* ($\mathcal{DIV}$consensus) problem [DSS98a], which may be solved with a slightly modified version of Chandra and Toueg's consensus algorithm [CT96]. In contrast with the traditional consensus problem, $\mathcal{DIV}$consensus participants are not required to have an initial value defined when starting the consensus algorithm; instead, the consensus algorithm asks the participant for its initial value when needed.[10] This property allows us to

[8] This may happen in case of a partition, if the sequencer is in a minority partition.

[9] We do not use a point-to-point invocation from the client to the sequencer because the client does *not* have to know which participant is the sequencer.

[10] Actually, the $\mathcal{DIV}$consensus algorithm proposed in [DSS98b] asks for an initial value only when the participant is the coordinator.

```
1: {Protocol of the client}
2: procedure OAR-multicast (m, dsts)        {Issue request for optimistic active replication}
3:    R-multicast (m, dsts) to dsts                {dsts is a set of processes (or a group)}
4:    wait until [for ⌈(|dsts|+1)/2⌉ processes q : received (reply_m, ts_q) from q with ts_q > 0
      or for any process q : received (reply_m, ts_q) from q with ts_q < 0]
                {The client waits for a majority of replies with the same positive timestamp,
                                                    or for any reply with a negative timestamp}

5: {Protocol of the server (code of process p)}
6: Initialization:
7:    order_p ← 1                                                    {Local message counter}
8:    ts_p ← 1                                                       {Local timestamp}
9:    dlvrs_p ← ∅                                  {Messages delivered since last consensus}
10:   s ← sequencer                                                 {Current sequencer}

11: when p = s and R-deliver (m)                {Sequencer process s orders the messages}
12:   send (m, order_s) to all
13:   reply ← OAR-deliver (m)                                       {Deliver and process request}
14:   send (reply, ts_s) to sender(m)                               {Send reply to sender}
15:   order_s ← order_s + 1                          {Increment the local message counter}

16: when p ≠ s and R-deliver (m)                {Non-sequencer process p waits for ordering
                                                                    information from the sequencer}
17:   wait until [received (m, order_m) from s or s ∈ D_p]          {Query the failure detector}
18:   if [received (m, order_m) from s] then                        {p received ordering information from s}
19:      dlvrs_p ← dlvrs_p ∪ {m, order_m}                           {dlvrs_p contains ordering information of
                                                                    all messages since last consensus}
20:      reply ← OAR-deliver (m)                                    {Deliver and process request}
21:      send (reply, ts_p) to sender(m)                            {Send reply to sender}
22:      order_p ← order_p + 1                          {Increment the local message counter}
23:   else                                                          {p suspects s to have crashed}
24:      propose (dlvrs_p)                                          {Launch consensus}
25:      wait until decide (d)
26:      while ∃n | (n, order_p) ∈ d do                             {Deliver messages starting from order_p,
                                                    in an order consistent with the replies sent to the client}
27:         n ← select one message such that (n, order_p) ∈ d       {(n, order_p) has been decided
                                                    by s, and thus all messages associated to order_p are identical}
28:         reply ← OAR-deliver (n)                                 {n is the next message to deliver}
29:         send (reply, −1) to sender(n)                           {Send reply to sender}
30:         order_p ← order_p + 1                          {Increment the local message counter}
31:      t ← largest order_n such that (n, order_n) ∈ d                          {t ← 0 if d = ∅}
32:      if t < order_p − 1 then                          {p has delivered more messages than those in d}
33:         resynchronize with primary partition
34:      dlvrs_p ← ∅                                    {Clear dlvrs_p upon consensus decision}
35:      ts_s ← ts_s + 1                                                {Increment timestamp}
36:      choose another sequencer
```

Figure 4.10: Optimistic Active Replication Algorithm

easily implement both active and semi-passive replication with the $\mathcal{DIV}$consensus problem.

The interaction model of the OGS consensus service is compliant with the $\mathcal{DIV}$consensus, since the consensus service calls back to the participants to obtain an initial value (see Section 3.2.2). Therefore, no modification is required to the IDL interfaces of the consensus service to use an algorithm based on the $\mathcal{DIV}$consensus.

Figure 4.11 illustrates a good run of the semi-passive replication algorithm given in

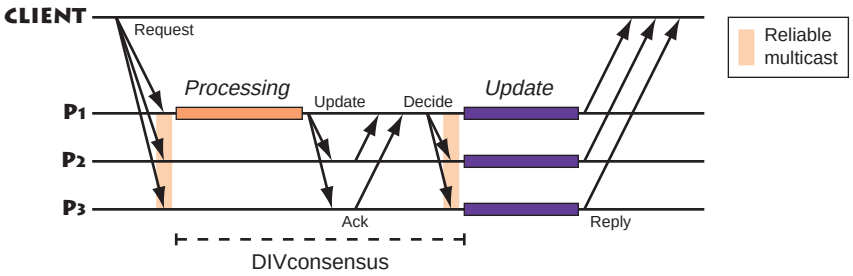

Figure 4.11: Using $\mathcal{DIV}$consensus for Semi-Passive Replication

Figure 4.12 (adapted from [DSS98a]). The primary processes the request when the consensus algorithm asks for an initial value. This processing must be performed without modifying the object's state (the state will be modified when processing the update). This initial value returned by the primary contains both a reply for the client and an update information for the backups. If there is no failure, the $\mathcal{DIV}$consensus algorithm decides on the value from the primary, and all backups update their state and send the replies to the client.

```
1: {Protocol of the client}
2: procedure SPR-multicast (m, dsts)          {Issue request for semi-passive replication}
3:    R-multicast (m, dsts) to dsts             {dsts is a set of processes (or a group)}

4: {Protocol of the server (code of process p)}
5: Initialization:
6:    handled_p ← ∅                                        {Requests already handled}
7:    unhandled_p ← ∅                                      {Requests not yet handled}
8:    cid_p ← 0                                                {Consensus identifier}

9: when R-deliver (m)                                  {Receive primary-backup request}
10:   if m ∉ handled_p then
11:      unhandled_p ← unhandled_p ∪ {m}

12: when getInitVal ()                                 {Return initial value for consensus}
13:   m ← select one request from unhandled_p
14:   {update, reply} ← handle(m)                {Process request without updating local state}
15:   return {m, update, reply}

16: when unhandled_p ≠ ∅                               {Handle primary-backup request}
17:   k ← cid_p
18:   DIVpropose (k, getInitVal)                                  {Launch DIVconsensus}
19:   wait until decide (k, {m_k, update_k, reply_k})
20:   send reply_k to sender(m_k)                                {Send reply to the client}
21:   update state according to update_k
22:   handled_p ← handled_p ∪ {m_k}
23:   unhandled_p ← unhandled_p \ {m_k}
24:   cid_p ← cid_p + 1
```

Figure 4.12: Semi-Passive Replication Algorithm [DSS98a]

4.4 Implementation Details

Several issues related to the implementation of OGS are worth discussing in the context of this dissertation. In particular, some implementation choices have been made for typed communication, group naming, concurrency management, client multicast protocol, and request duplication. These issues are presented in this section.

4.4.1 Typed communication

With the typed invocation interface of OGS, clients invoke operations directly on the server's interface, and OGS delivers multicast invocations by directly invoking the relevant operation of the server. There are two main solutions for implementing typed communication: smart stubs and dynamic ORB interfaces.

The first approach uses smart proxies on the client side and smart skeletons on the server side. These stubs hide groups from the application, and redirect requests and replies to OGS. They can be automatically generated from an IDL file. Although this approach is type secure and efficient — it uses static typing — it requires a special IDL compiler, and is both language and ORB dependent.

The second approach, adopted by OGS, uses two advanced features of the CORBA specification: the *Dynamic Skeleton Interface* (DSI) and the *Dynamic Invocation Interface* (DII). The DSI is used by the service to accept requests that actually aim at the server interface. The DII is used to construct the invocations for the server interface. We detail below how these features have been used to provide type transparency.

General Principle. The OGS approach to typed communication is similar to the CORBA *request level bridging* [OMG98a]. Translation from a client request to a multicast (for a set of servers) is performed by application style code outside the ORB. Client and servers mediate through a common protocol between distinct execution environments (possibly different ORBs).

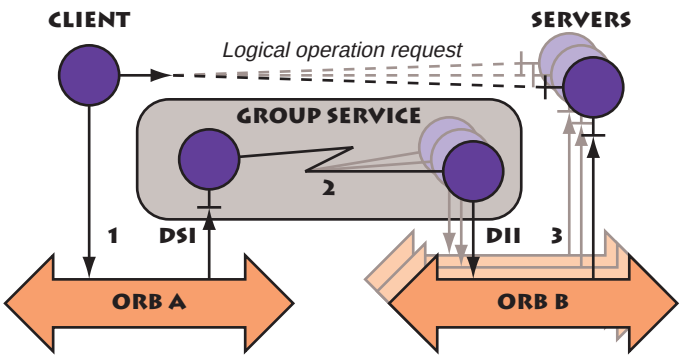

Figure 4.13: Providing Type Transparency

The general principle for implementing typed communication is illustrated in Figure 4.13: (1) the original request is passed to an OGS object (group accessor) in the client ORB: this object acts as a proxy for the servers; (2) the proxy object translates the request contents to an agreed format and issues a multicast to the server ORBs using untyped communication; (3) OGS server-side objects (group administrators) receive the multicast and invoke the required operation on the servers. Any operation result is passed back to the client using point-to-point communication. In comparison with untyped communication, this process requires two additional steps (1 and 3). In the following, we describe how these steps are implemented.

Accepting Requests Using the DSI: the Client Side. OGS gives the illusion to clients that they are directly invoking a server, whereas they are actually invoking the service. This greatly simplifies client development, since the latter can issue standard invocations to IDL-defined operations. OGS uses the DSI to accept any operation of the server interface, although this interface is not known at compile time. OGS intercepts invocations using a dynamic skeleton on the client side, transforms them into an agreed format, and multicasts them using untyped communication to all the group members. On the client side, OGS uses the CORBA *Interface Repository* (IR) for getting runtime information about the IDL interface of the replicated object.

When performing a typed multicast invocation, the service waits for a single reply from the servers, unless the operation is explicitly declared as one-way in the server's IDL interface, in which case a one-way multicast invocation is used.

Constructing Requests Using the DII: the Server Side. The DII allows an application to issue requests for any interface, even if this interface is unknown at compile time. OGS objects on server hosts receive the details of the request to be made as part of the multicast message sent by the client. The message contains information on the object that must be invoked, the operation name, the parameters, etc. OGS translates this into a DII call leading to the invocation of the requested operation on the server interface. Once the operation returns, OGS transmits the reply (or an exception that may have been raised by the application) back to the client.

4.4.2 Group Naming

Since group communication in OGS provides view change notifications, objects that are part of a group always know the current composition of their group. On the other hand, when an object wants to access a group of which it is not a member (e.g., to multicast a message or to join the group), OGS must first obtain a reference to the members of the group. This is achieved using *group names*, which are system-wide unique identifiers.

For managing group names, we use the CORBA *Naming Service* (NS) specified in [OMG97]. This service maintains name-to-object mappings in a federated ar-

chitecture. These name-to-object associations are called *name bindings*. A name binding is defined relative to a *naming context*, which is a CORBA object responsible for maintaining a set of bindings with unique names. Different names can be bound to an object in the same or different contexts at the same time. Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a naming graph. OGS uses one context per group, and stores references to individual group members and to service objects (e.g., group administrators) in these contexts. Figure 4.14 shows a naming graph with a *Bank* context that contains references to the two members of an object group (the dark circles represent naming contexts).

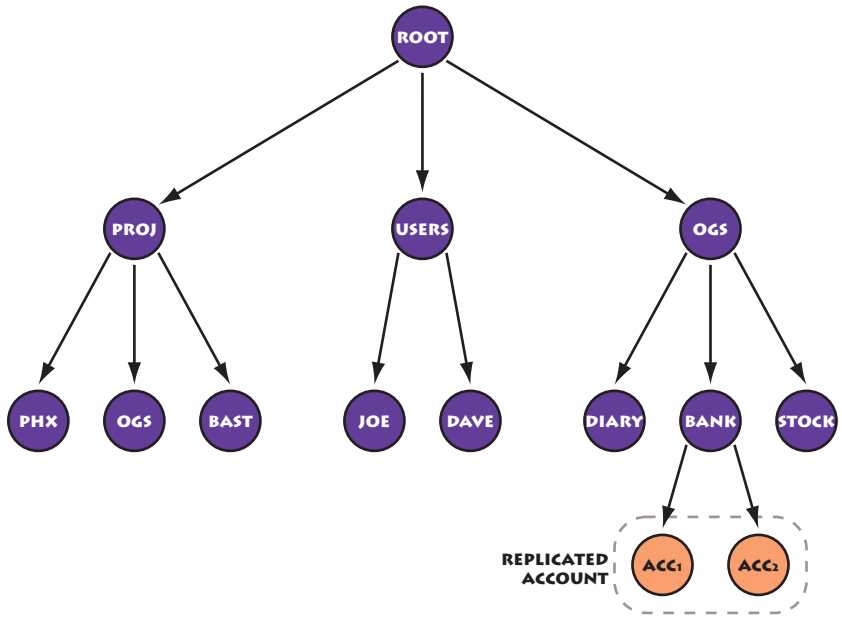

Figure 4.14: Sample Naming Graph with References to an Object Group

Although the naming service is not fault tolerant in itself, existing implementations are robust enough for our needs. In the worst case, a failure of the naming service can hinder a client to bind a group, but it will not disrupt objects that are already bound to the group. Alternatively, on could use a fault tolerant implementation of the naming service, such as the one described in [Maf96].

Keeping the naming service up-to-date is a difficult task. OGS updates the information in the naming service each time a view change occurs, but this information is not guaranteed to be up-to-date. In particular, if the last member of a group fails, OGS does not guarantee that the reference to the member will be removed from the naming service.

In practice, when an object wants to access a group, OGS contacts the naming service, retrieves the list of the group members, and contacts these members. Having only one of them being an actual member of the group is sufficient; if this were not the case, OGS would continue consulting the naming service until it succeeds in

contacting a group member.

4.4.3 Client Multicast Protocol

Whereas group members always know the exact composition of their group, clients may not have an up-to-date view of groups they are communicating with. The OGS implementation uses the following mechanism to ensure that multicast invocations reach all group members despite outdated client views.

Views have version numbers that are incremented every time the composition of the group changes. Upon creation, a group accessor obtains a list of all group members from the naming service, and stores this list in a view with a null version number. Each time a client issues a request to a group, the group accessor sends the request to all members of the current view, and embeds the version number of its local view in the request. Upon message reception, the members may decide to re-issue the request in the group depending on its semantics and the accuracy of the client's view (if the client view has a smaller version number than the one of the members, the chances are that the client did not send the request to all members of the current view). If the client's view is outdated, the first member of the group sends a new view to the client together with the reply.

Thus, clients have up-to-date views if they frequently send requests to a group. If the client's view is so outdated that no object of its view is currently a member of the group, the group accessor has to (transparently) obtain a fresh view from the naming service.

An interesting feature of the OGS client multicast protocol is that the client has merely to send a request to all members of its view, without caring about the semantics of the request. The client-side protocol is *identical* for all algorithms.

4.4.4 Concurrency Management

The OGS implementation assumes that it is running in a multi-threaded environment. Multi-threaded CORBA servers have the ability to process several requests at the same time. When dealing with requests that require a long processing time (e.g., database queries), multi-threading support is necessary to serve incoming requests while performing time consuming processing.

In OGS, multi-threading is especially important in one situation: when a client issues a multicast request to a group accessor, the latter launches a distributed protocol and waits for replies from servers before returning. In other words, multicasting is a *blocking* operation that expects replies (i.e., incoming messages) while being processed. Although this problem can be solved in a single-threaded system if the ORB implementation supports reentrant invocations, multi-threading provides a more general and powerful model.

Most ORB implementations provide support for forking new threads as a result of incoming requests. The main problem related to multi-threading is mutual ex-

clusion of shared resources. Since all threads of a process share the same data, the programmer must ensure that two concurrent threads do not modify the same critical resource at the same time, leading to inconsistencies. In particular, OGS provides ordering of requests, but the threads scheduler could break this ordering in the invokee process. In other words, an invocation i_1 could be delivered to the server object before another invocation i_2, but, since they execute in two different threads, i_2 could be processed *before* i_1. Therefore, special attention must be given to the problem of request sequencing.

In the OGS implementation, each incoming request executes in a separate thread, and the threads are sequenced. This scheme preserves ordering, at the price of potentially reduced performance. Thread sequencing is implemented using locking mechanisms based on the standard POSIX thread library *(pthread)* [NBF96]. Unfortunately, these locking mechanisms do not ensure FIFO ordering of waiting threads: if several threads block on some lock, the first thread to be unblocked will not necessarily be the first one that blocked. This limitation forced us to deal with explicit queues of threads, which results in a noticeable performance penalty.

4.4.5 Request Duplication

When a client invokes an actively replicated server, each replica receives the invocation, processes the requested operation, and returns a reply. There is no coordination between the replicas which behave as if they were not replicated. Active replication is not straightforward to achieve in an object-based environment like CORBA when an object may alternatively play the role of client and server. In fact, a replicated object can act as a client for another server, and so result in duplicated invocations (see the discussion on request filtering in Section 2.4) [GFGM98].

Filtering duplicated requests is a difficult problem. If a singleton object (which is not aware of OGS) is invoked by a client group, it is not possible to filter the request on the server side without some support from the ORB. On the other hand, filtering on the client side requires intercepting *every* outgoing request; this filtering can be done only if the request is issued via OGS. In the OGS model, it is possible to filter duplicated requests that are issued by a group to another group, but no support is provided for filtering requests issued to a singleton object.

A clean solution to the problem of duplicated requests would be to transparently associate a context — which may be a unique request identifier — to each request, and to forward this context along through chained invocations. This context could be used to detect duplicated requests and filter them. This solution requires some ORB support which is not currently available. With the current version of OGS, one would prefer to use primary-backup instead of active replication to avoid the problem of request duplication.[11]

[11] Notice that request duplication may also happen with primary-backup replication if the primary fails while processing a request.

4.5 Porting Experiences

Portability is a crucial aspect in the development of large scale distributed applications that use heterogeneous components. The degree to which CORBA guarantees portability of an application depends however on the nature of the application components. When these components rely on some features not completely specified by CORBA, portability may be affected. This is the case of OGS which relies on features such as non-blocking reliable communication, dynamic invocations, multi-threading, and predictable request scheduling.

We implemented OGS first using Orbix [ION97] and then we ported it to VisiBroker [Vis98a]. During this porting phase, we experienced several difficulties that are worth mentioning, and that shed some light on whether the current CORBA specification is mature enough to be deployed in systems that have portability requirements [FGS98a]. We also discuss problems resulting from deficiencies in the CORBA specification that are not limited to portability.

Multi-Threading

As mentioned before, OGS uses multi-threading to asynchronously wait for incoming events and perform background tasks like failure detection. Multi-threading is supported by all major operating systems but, as of version 2.1 of the CORBA specification, thread support and management are not specified, and thus are not portable. Although both Orbix and VisiBroker support multi-threading, they provide very different programming models. Both provide object-oriented wrappers for threads, locks, and condition variables that allow platform independence (e.g., between Posix and Windows NT threads), but these wrapper classes are not compatible with each other. This problem did not arise with the Java version of OGS, because Java provides language-level support for threads [Lea97].

The biggest difference between Orbix and VisiBroker threading models appears at thread creation: Orbix provides fine control over thread creation because the programmer explicitly creates threads in so-called *Filter* objects invoked before request processing. On the other hand, VisiBroker automatically and transparently starts threads for incoming requests, without letting the programmer choose which request has to be started in a new thread, and which does not. The VisiBroker model is easier to use and more efficient, but less powerful. OGS uses its own wrappers for encapsulating the differences between both models.

Server-Side Mapping

Although the CORBA specification suggests a C++ server-side mapping, it does not enforce implementations to comply with it. In particular, two models of skeletons are suggested: using inheritance and using delegation (*TIE* approach). Inheritance requires object implementations to inherit from an IDL-generated skeleton class, while delegation lets the skeleton forward requests to the object implementation,

without requiring any inheritance relationship.

Orbix and VisiBroker support both models, but use a different syntax and terminology. For instance, with the inheritance model, the skeleton class of interface `M::I` is called `M::IBOAImpl` with Orbix, and `_sk_M::_sk_I` with VisiBroker.

Another problem we faced is the generation by Orbix of an extra parameter for each IDL operation, of type `CORBA::Environment`. This parameter is used principally for signaling exceptions to the client with compilers that do not support native C++ exceptions, and is allowed by the CORBA specification. Since VisiBroker does not add this parameter, we had to use conditional compilation for each IDL-defined operation. In fact, much of the server-side mapping and BOA is underspecified. Writing portable code for OGS required extensive use of preprocessor macros.

Scoped Names

The CORBA specification is loose concerning the C++ mapping of scoped names, which can map to namespaces, nested classes, or concatenated identifiers. For instance, if an interface `I` is defined in a module `M`, an implementation can map `I` to a class in the `M` namespace, or to a nested class in the `M` class (`M::I`), or to the unnested class `M_I`.

This lack of precision is meaningful for compilers that do not support the latest C++ features, but it makes user code depend on the implementation choice of ORB vendors, and prevents developers from writing code which is portable to ORBs that implement different mappings. OGS assumes that the ORB supports nested classes.

Multiple-Inheritance of Implementations

When an IDL interface inherits from another interface, one would like to reuse the implementation of the base interface. In an object-oriented language like C++ that supports multiple inheritance, the implementation of the derived IDL interface can inherit from both the implementation of the base interface and from the skeleton of the derived interface. This programming model works fine with Orbix, but is not fully supported by VisiBroker. In OGS, we had to change our implementation to use delegation instead of multiple inheritance.

Object Equivalence

Another problem relates to the lack of clear semantics concerning object reference comparison, which is often required in a distributed application where several object references must be checked for equality. CORBA only specifies an operation, called `_is_equivalent()`, that compares two object references; it returns `true` if they designate the same object, and `false` if the underlying ORB cannot determine whether the object references are equivalent or not, which often happens when dealing with references from different ORBs. Therefore, a compliant implementation may always return `false`, which is obviously useless. A workaround for this problem is to define

an explicit operation on the objects that need to be compared, and to invoke it for each comparison; but this involves a remote invocation for each object reference comparison. In general, it is better to design applications so that they do not need to test for object equality, but this is not always possible. OGS tests object equality in rare situations, and relies on `_is_equivalent()`.

Any Type Mapping

Values of type `any` are used intensively in OGS, but VisiBroker does not comply with the CORBA specification concerning the C++ mapping of the `any` type. For extraction of complex data types, such as structures, the specification states that the extraction function must be prototyped for passing parameters as pointers rather than values to increase efficiency. As VisiBroker does not use pointers, the application developer cannot write portable code.

Dynamic Exception Handling

The typed version of OGS uses the DSI on the client side to intercept requests performed on the server's interface, and the DII on the server side to reconstruct the request and invoke the servers. In between, the request is transmitted and the reply is returned by OGS as untyped values, i.e., they are packed in CORBA `any` variables. Although it is easy to insert the request's parameters and return values into `any` variables, the CORBA specification does not make it possible to pass back all types of exceptions from the server to the client. On the client side, returning an exception resulting from a DSI invocation requires giving an `any` value containing the exception to the dynamic request object. On the server side, an exception returned as a result from the DII invocation is a pointer to an object deriving from the `Exception` class, and there is no way of packing this object into an `any` variable.[12] This implies that the typed version of OGS does not allow the return of all types of exceptions at the moment.

Initial Services

CORBA applications can list available services using the `list_initial_services()` function, and they can obtain an initial reference to one of these services using the `resolve_initial_references()` function. This makes it easy for applications to use CORBA services in a portable way. Unfortunately, installing a new service in the list of initial services is implementation-specific, and there is no way to add user-defined services. Therefore, getting an initial reference to OGS is not possible using these functions. A solution is to use the *Object Trader Service*, but very few implementations of this service are currently available. The solution we adopted is to simply register OGS in the Naming Service.

[12] Unless the resulting exception is of type `UnknownUserException` which is not supported by Orbix.

4.6 Performance

This section presents the performance of the OGS implementation on VisiBroker [Vis98a] (some performance measurements of OGS on Orbix [ION97] may be found in [FGS98b]).[13] We focus on client multicast invocations, i.e., the cost of invocations through OGS, from a client to a group of objects. We compute the overhead of using OGS compared to using plain CORBA invocations, and we investigate the sources of this overhead.

These measurements focus on OGS performance. They do not present the intrinsic cost of invocations going through the ORB in detail. Exhaustive performance measurements of CORBA latency with different ORBs may be found in [GS98].

4.6.1 System Configuration

Our performance measurements have been performed with the C++ version of OGS, compiled with VisiBroker 3.2. Testing took place on a local 10 Mbit Ethernet network, interconnecting 13 Sun SPARCstations running Solaris 2.5.1 or 2.6, under normal load conditions (all workstations were running X Windows, as well as several user applications such as netscape or emacs). Among these workstations, there were four Sun UltraSPARC 30 (250 Mhz processor, 128 MB of RAM), and nine Sun UltraSPARC 1 (170 Mhz processor, 64 MB of RAM). For tests involving up to four hosts, only the UltraSPARC 30 workstations were used. All the client and server applications were located on different hosts, except the OGS daemon which was located on the same host as the client. The tests have been run with the *TCP_NODELAY* option that sets all sockets to immediately send requests, instead of buffering them and sending them in batches.

4.6.2 Test Scenarios

Our performance tests evaluate the latency of multicast invocations issued by a client to an object group when no failure occurs. These invocations use the various semantics provided by OGS: total order (consensus-based and sequencer-based), reliable, and unreliable; and three different modes of invocations (Figure 4.15): untyped invocations with the OGS library, untyped invocations with the OGS daemon, and typed invocations with the OGS daemon (execution models are discussed in Section 5.1.1). The group size varies from one to ten members. The client waits for a single reply from the servers, except with the optimistic active replication algorithm, with which the client waits for a majority of replies. The arrows in the figure represent the invocation path followed by the requests and the replies.

The test program operates as follows: a single client executes several rounds, in each of which it issues a fixed number of synchronous invocations (typically 100). The client waits for a reply from each request before issuing the next invocation. The

[13] Due to some limitations in the current version of Orbix, we could not perform all the tests presented in this section with the Orbix version of OGS.

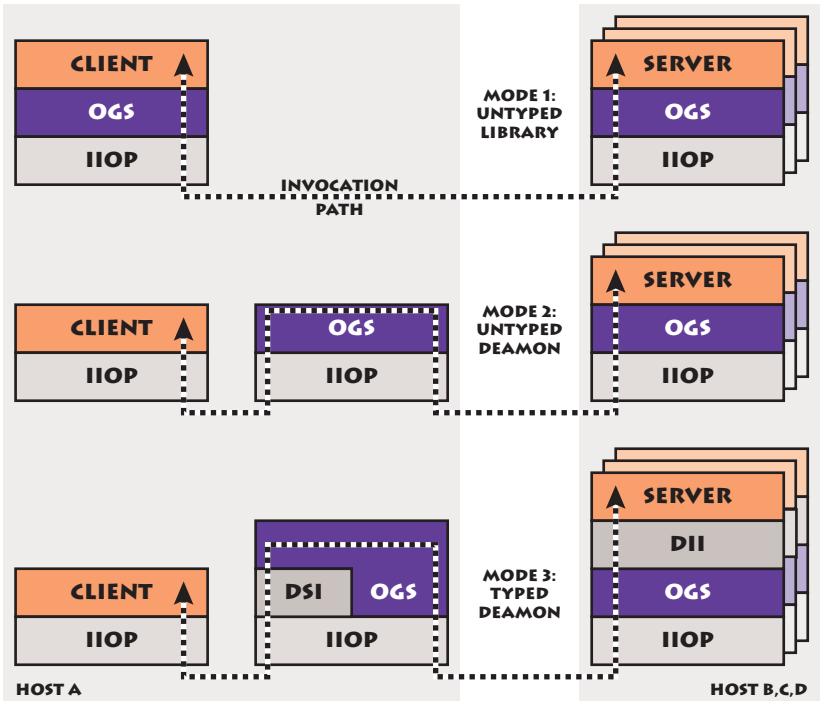

Figure 4.15: Three Test Models for Client Multicast Invocations

total time of each round is divided by the number of invocations issued during the round to obtain the latency of a single invocation. We kept the value of the best round. Since there is only one client, invocations are not performed concurrently and OGS cannot benefit from its consensus-based total order algorithm that can order several requests at once. Therefore, this test is not a good measure of the total throughput of OGS with this algorithm. The results of the program execution are given in Table 4.1. These results are discussed in the next section.

4.6.3 Evaluation

In Section 2.4, we have classified transparency according to three categories: behavior transparency, plurality transparency, and type transparency. We analyze the costs of the OGS architecture according to this classification: we first evaluate the cost of the various multicast primitives of OGS (behavior); then, we evaluate how much the performance depends on the group size, and how it compares to a single invocation through the ORB (plurality); we finally evaluate the performance overhead induced by the use of dynamic typing facilities in OGS (type).

The Cost of Behavior

The semantics of multicast invocations depend on the behavior of an object group. Reliability and speed are often antithetical. Figure 4.16 illustrates the cost of the dif-

Exec. style	*Semantics*	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Untyped OGS Library	**Total Order**	2.80	7.57	9.61	13.65	18.81	28.07	38.33	47.42	60.97	79.80
	Opt. Active Repl.	2.77	4.74	4.98	5.92	6.58	8.15	8.97	10.42	11.49	13.01
	Reliable	2.70	3.03	3.36	4.72	7.82	13.28	18.22	23.00	27.83	34.60
	Unreliable	2.70	2.90	3.14	3.49	4.24	5.11	5.75	6.46	7.67	8.67
Untyped OGS Daemon	**Total Order**	3.78	8.60	10.51	14.80	19.91	29.75	39.50	51.02	63.18	82.81
	Opt. Active Repl.	3.69	5.89	6.29	7.19	7.51	9.17	10.60	11.91	12.22	13.18
	Reliable	3.66	4.14	4.17	6.72	9.32	14.12	19.91	24.60	28.35	36.81
	Unreliable	3.69	4.01	4.05	4.89	5.29	5.90	7.07	7.86	9.11	10.05
Typed OGS Daemon	**Total Order**	23.56	29.75	33.18	36.64	45.15	52.00	64.67	76.62	92.51	107.27
	Opt. Active Repl.	23.48	24.73	29.65	31.19	32.69	33.86	37.14	42.13	45.15	49.73
	Reliable	23.24	24.88	28.61	30.38	34.67	39.92	43.21	51.19	60.97	70.99
	Unreliable	23.33	24.07	26.83	27.79	30.92	32.13	35.65	39.95	43.21	47.15
ORB	**Unreliable**	0.88									

Table 4.1: Performance of Multicast Invocations with Various Group Sizes and Execution Styles (ms./inv.)

ferent OGS untyped invocation primitives, with the library execution style (mode 1 in Figure 4.15) and different group sizes.

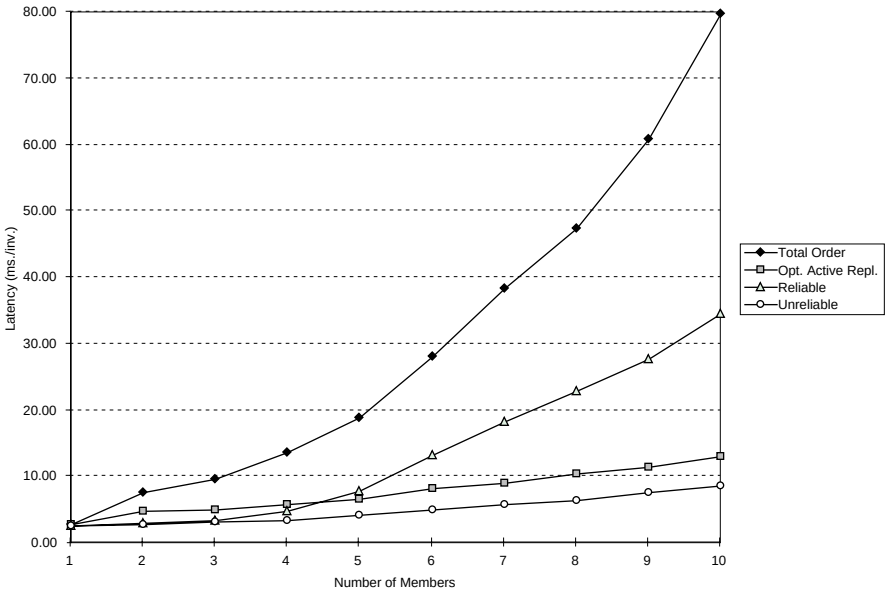

Figure 4.16: Comparing OGS Multicast Primitives

This figure shows that the total order and reliable multicast primitives grow faster than the other primitives. This is due to the fact that the former primitives are based on the simple reliable multicast algorithm of Section 4.3.3; the complexity of the number of messages for this algorithm is $O(n^2)$ for n participants. This cost could be reduced by using another reliable multicast algorithm (such as the one suggested in Section 4.3.3). In contrast, the optimistic active replication algorithm has been optimized so that is does not use a reliable multicast primitive, as described in Section 4.3.5. Its cost grows thus linearly, similarly to unreliable multicast.

The Cost of Plurality

In comparison to the invocation of a single object, invoking a group of objects involves communicating with multiple objects. The plurality introduced by object groups has a cost. Figure 4.17 compares the latency of invocations issued through OGS, with a corresponding invocation issued directly through the ORB. It illustrates the cost of plurality (i.e., the additional cost induced by the addition of members to the group), as well as the cost of the service (i.e., the cost of the extra indirection, and of parameter marshaling and unmarshaling). OGS invocations are performed using the library execution style, with untyped invocation semantics. The invocation sent through the ORB is a standard two-way request issued through static stubs and skeletons.

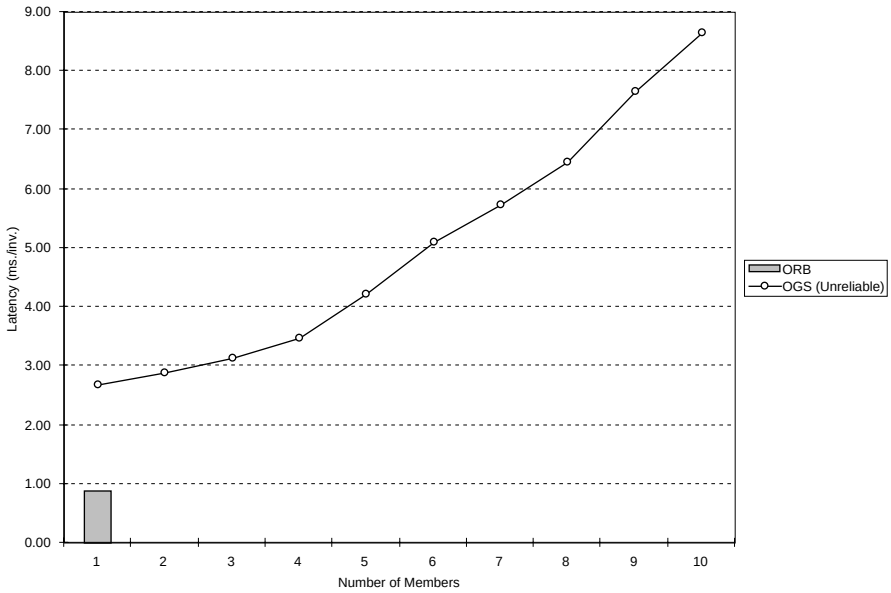

Figure 4.17: The Cost of Plurality in OGS

Figure 4.17 shows that the cost of passing through the service is slightly less than 2 milliseconds per invocation, and is *three times* higher than the cost of a standard invocation through the ORB. This overhead is fixed and does not depend on the number of participants: sending an unreliable request to two objects is *not* twice slower than invoking a single object.

The fixed cost of OGS consists basically of the following actions: OGS accepts client requests, and builds a message that it multicasts to all servers;[14] on the server-side, OGS extracts the data from the message, passes it to the server, gets the return value, and builds the reply message for the client; when the first replies arrives, OGS extracts the result and returns it to the client. With more complex communication protocols, this cost becomes comparatively negligible.

[14]Notice that OGS uses one-way messages for sending unreliable requests.

The Cost of Typing

The cost of typing appears essentially at two places in the OGS architecture: when using the typed version of OGS, and when managing untyped values of type `any`.

Untyped vs. Typed Invocations. Type transparency is an important feature of OGS because it hides groups from the application developer, and makes it possible to reuse existing applications without having to modify the client. In the current version of OGS, typed communication is available only for the daemon execution style. Figure 4.18 compares the latency of untyped totally ordered requests (library and daemon execution styles) with that of typed requests.

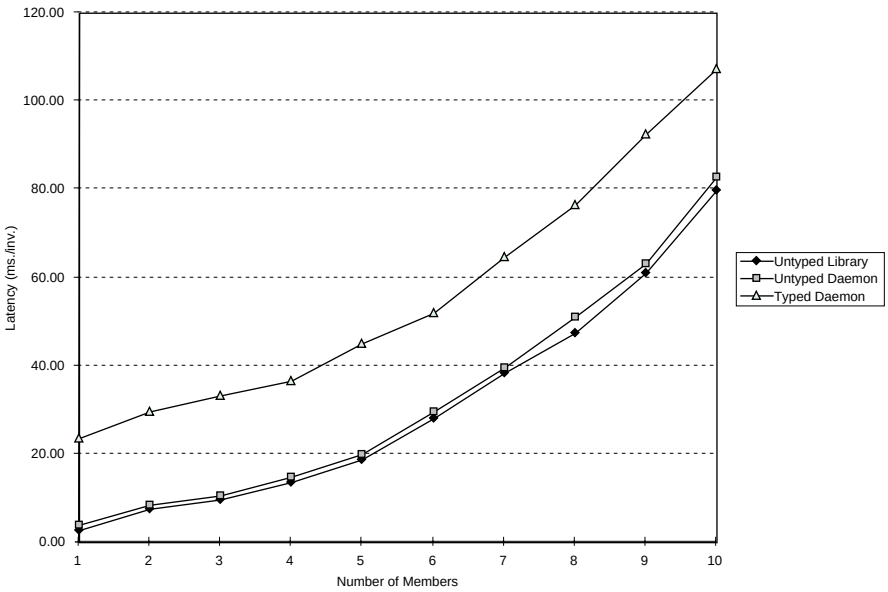

Figure 4.18: Untyped vs. Typed Communication

This figure illustrates that there is a fixed overhead of about 1 millisecond for using the daemon. This corresponds to the latency of a single two-way invocation through the ORB. The typed version of OGS adds an overhead of about 20 milliseconds. This overhead results from the use of the DSI and the DII for type transparency, and is *independent* of the group size. This is due to the fact that the DSI and the DII are used only once on the client and the server side.

Request Management. When profiling OGS, we noticed that a non-negligible part of the time required for remote invocations is spent in constructing requests. We also observed that working with untyped `any` values has a significant impact on performance. Unlike other IDL types, `any` values are augmented by a `typecode` information that contains details about the actual type of the value. This information increases the size of the messages sent on the network. Moreover, validity checks upon data extraction slow down the remote invocation process.

Table 4.2 presents the cost associated to the management of untyped `any` values with VisiBroker, for (1) inserting simple and complex types into `any` variables, (2) copying `any` variables, and (3) extracting simple and complex types from `any` variables. The simple data type is a `long` variable; the complex data type is a structure composed of a `long` variable, and a sequence of object references containing three elements, each of which designates the root context of the naming service. Results are expressed in operations per millisecond, and are the average of several thousands of executions.

Operation	*#op./ms.*
Insertion (simple type)	401.33
Insertion (complex type)	6.37
Copy (simple type)	354.773
Copy (complex type)	7.63
Extraction (simple type)	6920.42
Extraction (complex type)	0.48

Table 4.2: Cost of Managing Untyped `any` Values (op./ms.)

Table 4.2 shows that operations on complex types are much more costly than those on simple types. In particular, the extraction of a complex type costs about 2 milliseconds; this is *twice* that for a remote invocation through the ORB. This cost is not negligible, and explains the price of the extra indirection through OGS.

⋆

⋆ ⋆

Summary

OGS has been implemented in C++ using Orbix 2.xMT [ION97], and then ported to VisiBroker for C++ 3.x [Vis98a] and VisiBroker for Java 3.x [Vis98b]. For this implementation, we have considered an asynchronous system augmented with an unreliable failure detection mechanism [CT96] (the OGS monitoring service), and we make the assumption that links are reliable. OGS uses exclusively the communication primitives provided by the ORB as a temporary substitute for OMG's future messaging service.

OGS uses *protocol interfaces* to exchange the messages required for executing distributed protocols. These interfaces are implementation-specific and are not accessible by the application. They reflect the dependencies between the different OGS components, by inheriting from interfaces of several components.

OGS implements several algorithms for group communication and membership. The consensus is based on the algorithm of Chandra and Toueg [CT96], and is used essentially for group membership. Reliable multicast is performed using a simple diffusion-based algorithm. Total ordering of messages is achieved by two distinct algorithms: the first one is based on the consensus service and is combined with the group membership algorithm; the second one is a new optimistic algorithm for active replication. Primary-backup replication is implemented using a consensus algorithm based on deferred initial values [DSS98a].

OGS implements typed communication using CORBA dynamic interface mechanisms: the *Dynamic Skeleton Interface* (DSI), the *Dynamic Invocation Interface* (DII), and the *Interface Repository* (IR). Group naming is implemented using the standard CORBA naming service [OMG97], which maintains name-to-object mappings in a federated architecture. Concurrency management is achieved using system-specific mechanisms (POSIX threads [NBF96]), because the CORBA specification does not standardize thread support.

When implementing and porting OGS, we discovered several limitations of the current CORBA specification. In particular, portability is a crucial aspect in the development of heterogeneous distributed applications; however, we could not develop a fully portable implementation of OGS due to deficiencies in the CORBA standard. Similarly, several issues are underspecified in the current CORBA specification, such as multi-threading, server-side mapping, multiple-inheritance of implementations, object equivalence, `any` type mapping, and dynamic exception handling.

Performance measurements of OGS show that the throughput for a single client is 100 group invocations per second with three members and total order multicast (based on the consensus algorithm), and 12 invocations per second with ten members. With the optimistic algorithm for active replication, the throughput is 200 invocation per second with three members and 75 with ten members. There is a fixed price to pay (about 20 milliseconds) when issuing a typed invocation.

Chapter 5

Programming with OGS

Which road do I take? she asked.
Where do you want to go? was his response.

L. Carroll

Many applications can benefit from group communication. Among them are applications that have fault tolerance and high-availability requirements (e.g., power plant control, financial applications), applications that need to preserve consistency between several components and to share information (e.g., collaborative authoring), applications that perform parallel request processing (e.g., database lookup, time-consuming computations), software life cycle, etc. Depending on the application and how it uses group communication, the interface and behavior of group members can be identical or different:

- Same interface, same behavior: *active replication* requires that all group members have the same interface, and that they have a deterministic behavior. All members perform the same task and handle all requests.
- Same interface, different behavior: with *primary-backup* replication or *load-balancing*, all members have the same interface, but perform different tasks. Only one member handles a request.
- Different interfaces, same behavior: groups can be used for *multi-versioning*. Members perform the same task, but do not have the same interface. They only share a common interface that provides compatibility between the different versions.
- Different interfaces, different behavior: *monitoring and control* is easy to implement using heterogeneous groups of objects that perform different tasks and do not have the same interfaces. Groups are used to detect member failures, although the tasks of individual group members are completely unrelated.

In this chapter, we focus on four applications of different types, and show how these applications can be implemented using OGS (with VisiBroker 3.x). For the sake of simplicity, all error-handling code has been removed from the examples presented in this chapter.

5.1 OGS Configuration

Invocations to object groups are performed by OGS. Clients messages are sent via group accessor objects, and server messages are delivered via group administrator objects. Group accessors and administrators are *service objects* that form the visible part of the OGS runtime system, which is presented in this section. The application developer can configure this runtime system in a number of ways, leading to different degrees of flexibility, efficiency, transparency, or reliability.

5.1.1 Execution Models

To conceal efficiency and flexibility, OGS provides two execution models: a *linkable* model and a *daemon* model. In the first model, the service objects are co-located with application objects, i.e., they are linked with the application and they execute in the same address space (or process). In the second model, the service objects are located in another process — the `OGSd` daemon program — which may be on the local or on a remote host.

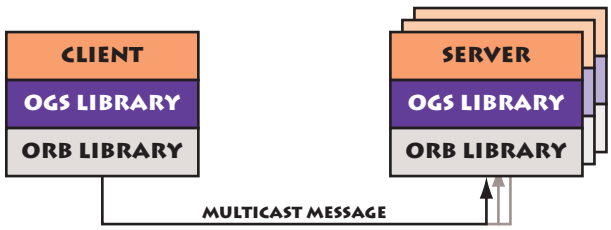

Figure 5.1: The Co-located Execution Model

The linkable version of OGS is provided as a C++ dynamic library (`OGSl`) to be linked with C++ applications, or as a set of Java classes usable from Java applications (see Figure 5.1). This execution model is more efficient since inter-process communications are more costly than invocations between objects located in the same process [GFGM98]. Nevertheless, it forces the code of the application to be written with the same programming language as the library and to support multi-threading.

The *daemon* execution model, with two separate processes, has the advantage of decoupling the service from the application, enabling several applications running on the same host to use the same resources. It also allows user applications written in another programming language, such as Smalltalk, to use the C++ or Java service.

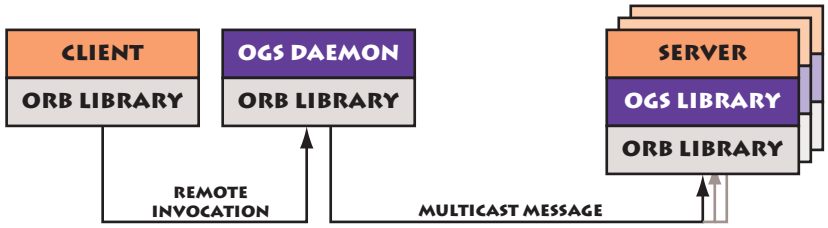

Figure 5.2: The Remote Execution Model

Figure 5.2 illustrates the use of the OGS daemon on the client side, while the service is linked with the application on the server side.[1]

The co-located execution model benefits mainly from its efficiency, while the remote execution model provides language heterogeneity. But the choice of the execution model also affects reliability, as described in the next section.

5.1.2 Service Location

Whereas CORBA objects should be independent of their real location, some objects of OGS have to be located on the client and server sites for the services to provide the required reliability. This is more a semantic requirement than an architectural requirement, since service objects can actually be installed anywhere.

With the co-located execution model, the application does not need to care about potential link failures or crashes of service objects, since the service cannot fail independently from the application. Therefore, the client sees the group as a highly available entity, and the group accessor masks failures of individual components.

If the application and the service objects are in two separate processes located on the same machine, only a crash of the OGS daemon process can prevent the application from using the service.

If the application and the service objects are located on different machines, the application must handle network, machine, and OGS daemon process failures. In this situation, when using groups for replication, a replicated server behaves exactly as a remote singleton object from the client's perspective. If the group accessor fails or a network partition occurs between the client and the group accessor, it has the same semantics as a failure of the singleton object: the client does not know whether the server failed before of after handling the request, and the client has to re-bind the server. The replicated server keeps its state consistent (i.e., all copies are kept identical) in spite of a failure of the group accessor, but the failure is not transparent to the client.

As already mentioned, the group reference is encapsulated in the group accessor, which acts as a client-side group representative. But this group reference can also be *shifted to the server side*: since group administrators inherit from group accessors,

[1] Notice that the servers could also use an OGS daemon.

the client can use a remote group administrator for invoking replicated servers. This model is somewhat similar to using a remote group accessor: if the server to which the client is bound fails, the client can react as in a non-replicated situation, i.e., try to re-bind the object. But it can also benefit from replication, and use any other administrator from the group.

Table 5.1 summarizes the different types of failures that may occur depending on the service location, and how the client has to react.

Fault handling	**Service objects are co-located**	**Service objects are on the same machine**	**Service objects are remote**
Service objects failures	Don't care	Restart and re-bind service	Restart and re-bind service
Link failures between application and service objects	Don't care	Don't care	Retry invocation
Member object failures	Don't care	Don't care	Don't care

Table 5.1: Fault Handling in OGS

5.1.3 Service Instantiation

A group accessor is an object that encapsulates the structure and behavior of a group reference, and that remembers and tracks the composition of the group. Ideally, it should be a temporary object, created on-the-fly when a group reference enters the address space of the application. However, in OGS it is not possible to create a group accessor implicitly without some ORB support. So the group accessor has to be created (i.e., *instantiated*) explicitly.

This creation may be performed directly by the client, using an object factory. With this approach, the client explicitly invokes the factory to create a new service object. Figure 5.3 illustrates how a client creates a group accessor. The client first binds to an object factory, and issues a request to create a group accessor (1, 2). The factory returns a reference to newly created object, that the client uses to invoke the object group (3, 4).

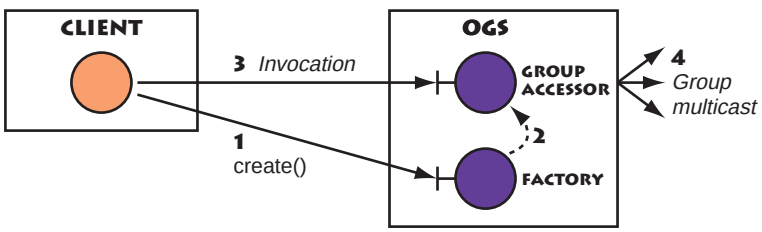

Figure 5.3: Explicit Service Instantiation

Creation may also be performed by a third party (such as the `ogsutil` program provided with OGS). The reference to the newly created service object may be given to the client through the naming service, for instance. Figure 5.4 presents

implicit service instantiation. A third party first creates a group accessor using an object factory (1, 2), and registers this accessor in the naming service (3). The client gets the reference to the group accessor from the naming service (4), and invokes the object group (5, 6).

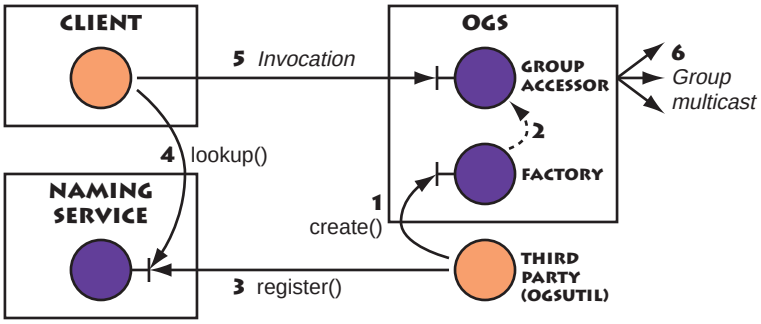

Figure 5.4: Implicit Service Instantiation

Explicit service instantiation requires the client to perform an *initial binding phase* to bind to the service and register with it. Once this binding phase has been performed, the client may invoke the server through OGS. This approach makes the use of OGS explicit, and requires the client to be aware of groups.

On the other side, implicit service instantiation does not require the client to be aware of OGS. When using this approach in combination with the typed version of OGS for replicating a server, the reference registered in the naming service bears the type of the server. This means that the client completely ignores that the server is replicated when accessing it through the reference from the naming service. That way, *full client transparency* is achieved, both for service instantiation and for replicated server's invocation. Table 5.2 summarizes the different degrees of transparency that the client application may achieve using OGS.

Transparency	**Application uses untyped OGS**	**Application uses typed OGS**
Application creates service objects	No transparency	Transparent invocation
Third party creates service objects	Transparent instantiation	Full transparency

Table 5.2: Transparency in OGS

5.2 Programming Methodology

OGS provides support for gathering CORBA server objects into logical groups and communicating atomically with them. To benefit from group communication, the program has to interact with OGS and provide application support for object groups. The programming methodology for OGS application development is presented in this section.

5.2.1 Developing without OGS

The easy way to program with object groups is to start by developing an application without OGS. Once the interfaces have been defined and the application has been implemented and tested without OGS, group support may be added with minor modifications to the application's code. An advantage of this methodology is that it allows to reuse existing applications and turn them fault tolerant *a posteriori*.

If groups are *not* used for replication, special care has to be taken when defining the IDL interfaces of the application. In particular, if the application wants to access all the replies resulting from an invocation to an object group (e.g., when using groups for parallel processing), the typed version of OGS cannot be used transparently: transparent invocations return only one reply to the client. In this situation, though, it is possible to preserve transparency by decoupling the application interfaces so that requests are issued through one-way multicast invocations, and replies are returned explicitly to the caller through point-to-point invocation (see example in Section 5.4).

5.2.2 Making Server Objects Groupable

Once the application runs well without OGS, the next step is to add group support to the server objects. This is performed by having servers inherit from the `Groupable` IDL interface. This interface defines several operations that the server objects have to implement to be member of a group. The following operations must be implemented by the application developer:

- *Support for message delivery:* messages sent using the untyped version of OGS are delivered to the server objects through their `deliver()` operation. If the application uses only the typed version of OGS, this operation may be left empty.

- *Support for view change notification:* when a new object joins a group, or a member object leaves or fails, all member objects are notified through their `view_change()` operation. They receive an ordered list of current group members, which may be used for instance to deterministically decide upon the role of each object in the group.

- *Support for state transfer:* when a new object joins a group, it atomically receives the shared state from the other members of the group. This is generally required to preserve the application consistency. The state transfer mechanism is implemented by two operations, `get_state()` and `set_state()`, that are respectively invoked on a current and on the new member.

Typical implementations of these operations are given in Sections 5.3, 5.4, and 5.5.

5.2.3 Service Instantiation: Server Side

A typical CORBA server creates one or several objects, exports them to the ORB, and waits for incoming events. When programming with OGS and explicit service instantiation, a few additional actions must be performed after the server object has been created (Figure 5.5):

- Before using OGS, the server has to bind to a group administrator factory, generally located on the same machine as the server (locality constraints are discussed in Section 5.1.2). The initial reference to the object factory may be obtained for instance from the CORBA naming service, or via a stringified reference given through the command line.
- The second step consists in creating a group administrator. This is done by invoking the `create()` operation of the group administrator factory, with a group name as parameter. A group administrator is always attached to one group.
- Finally, the groupable server object can be added to the group by invoking the `join()` operation of the group administrator. This eventually leads to a view change notification, indicating that the groupable object is now member of the group.

```
1  // C++
2  int main(int argc, char *argv[])
3  {
4    // Initialize the ORB
5    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
6    CORBA::BOA_var boa = orb->BOA_init(argc, argv);
7
8    // Create groupable application object
9    MyGroupable_var server = new MyGroupable_i();
10   boa->obj_is_ready(server);
11
12   // Service instantiation
13   // 1) Bind to a group administrator factory
14   mGroupAccess::GroupAdministrator_var gaf = ...;
15
16   // 2) Create a group administrator
17   mGroupAdmin::GroupAdministrator_var ga = gaf->create(argv[1]);
18
19   // 3) Add server to the group
20   mGroupAdmin::InterfaceSemantics sem;
21   sem.default_semantics_ = mGroupAccess::TOTAL_ORDER;
22   ga->join_group(server, sem);
23
24   // Export the server to the network
25   boa->impl_is_ready();
26 }
```

Figure 5.5: Service Instantiation of a Typical OGS Server (C++)

When joining a group, the server object can specify the semantics associated with each operation of its interface, and that will be used by the typed version of OGS.[2] This is performed by passing a structure of type `InterfaceSemantics` to the `join()` operation. After these steps have been completed, the server application waits for incoming events, as any other CORBA server.

5.2.4 Service Instantiation: Client Side

Unlike the server, the client application does not need to provide OGS support through IDL interfaces. The client simply uses OGS service objects for communicating with object groups. A client typically performs the following actions when using implicit service instantiation (Figure 5.6):

```
1  // C++
2  int main(int argc, char *argv[])
3  {
4    // Initialize the ORB
5    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
6    CORBA::BOA_var boa = orb->BOA_init(argc, argv);
7
8    // Service instantiation
9    // 1) Bind to a group accessor factory
10   mGroupAccess::GroupAccessorFactory_var gaf = ...;
11
12   // 2) Create a group accessor
13   mGroupAdmin::GroupAccessor_var ga = gaf->create(argv[1]);
14
15   // 3) Cast group accessor
16   CORBA::Contained_var cv = ir->lookup("MyGroupable");
17   CORBA::InterfaceDef_var in_def = CORBA::InterfaceDef::_narrow(cv);
18   MyGroupable_var server = MyGroupable::_narrow(ga->cast(in_def));
19
20   // 4) Invoke server
21   server->operation();
22 }
```

Figure 5.6: Service Instantiation of a Typical OGS Client (C++)

- The first two steps are similar to the server application. The client binds to a group accessor factory, and creates a group accessor by invoking the `create()` operation of the factory, with a group name as parameter. A group accessor is always attached to one group.
- The client can then optionally use the `cast()` operation of the group accessor to obtain a reference to a *typed* group accessor object that supports the same interface as the server.
- Finally, the client can invoke the object group using typed or untyped communication.

[2] If all servers of a group do not specify the same semantics for an operation, there is a risk of inconsistency. OGS does not currently deal with this situation.

Note that all these actions may be ignored when using implicit service instantiation, as described in Section 5.1.3.

5.3 Replication with OGS

Our example of a highly available replicated application is a simplified distributed bank account. The account maintains a current balance, and clients perform operations on the account by *depositing* and *withdrawing* money. Withdrawals may be performed only if there is enough money in the account. Some operation pairs are commutative (e.g., *deposit* is commutative with itself), while others are not (e.g., *deposit* and *withdraw* are not commutative in some situations), requiring total ordering.

5.3.1 Design

The application is composed of two parts: the servers that perform computations and the clients that perform operations on the replicated account. The shared state of the servers is the account's current balance.

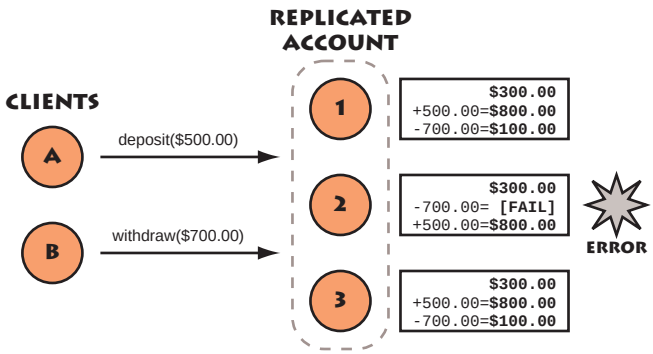

Figure 5.7: Account Example Application

Figure 5.7 presents a typical configuration with three copies of a replicated account, and two clients accessing the account concurrently. In this scenario, client A performs a deposit of \$500 and another client B withdraws \$700. These operations are not commutative, and it turns out that servers 1 and 3 order the deposit before the interest computation, while server 2 does not. The withdrawal cannot be performed on server 2 because there is not enough money in the account. This scenario illustrates a loss of consistency due to wrong request ordering: the request should be totally ordered to avoid this situation. OGS transparently preserves system consistency by delivering requests in the same order to all replicas.

5.3.2 IDL Specification

The IDL interface of the account application is given in Figure 5.8. It simply consists of an `Account` interface, with one attribute and two operations. The `balance` attribute (line 3) represents the current amount of money in the account, and should not be modified directly; therefore, it is declared as *read-only*. The `deposit()` and `withdraw()` operations (lines 5–6) modify the account's balance and return the amount of money that was effectively added to or removed from the account.[3] The account's interface inherits from OGS' `Groupable` interface (line 2) to allow replication.

```
1  // IDL
2  interface Account : mGroupAdmin::Groupable {
3    readonly attribute float balance;
4
5    float deposit(in float amount);
6    float withdraw(in float amount);
7  };
```

Figure 5.8: IDL Interfaces of the Account Server

5.3.3 Server Implementation

```
1  // C++
2  class Account_i : _sk_Account
3  {
4    CORBA::Float balance_;
5
6  public:
7    Account_i() : balance_(0.0) {}
8
9    // Groupable IDL operations
10   virtual CORBA::Any* get_state();
11   virtual void set_state(const CORBA::Any& state);
12   // ...
13
14   // Account IDL operations
15   virtual CORBA::Float balance()
16   { return balance_; }
17   virtual CORBA::Float deposit(CORBA::Float amount)
18   { balance_ += amount; return amount; }
19   virtual CORBA::Float withdraw(CORBA::Float amount)
20   { if(amount > balance_) return 0.0;
21     balance_ -= amount; return amount; }
22 };
```

Figure 5.9: C++ Interfaces of the Account Server

[3] If a withdrawal cannot be performed because of an overdraft, `withdraw()` returns 0. In this situation, a cleaner solution would be to use an exception.

The C++ interface of the account server is shown in Figure 5.9. The implementation of the operations are inlined in the C++ interface definition. To keep the code simple, no check is performed for negative amounts or balance overflow. The current balance of the account is kept in the member's `balance_` variable.

The account server also has to implement operations from the `Groupable` interface (Figure 5.10). In particular, it has to provide support for state transfer (`get_state` and `set_state`), and tell the service the semantics associated to each operation.

```
1  // C++
2  CORBA::Any* Account_i::get_state ()
3  {
4    // Pack the state into an any
5    CORBA::Any* a = new CORBA::Any();
6    *a <<= balance_;
7    return a;
8  }
9
10 void Account_i::set_state (const CORBA::Any& a)
11 {
12   // Extract the state from an any
13   a >>= balance_;
14 }
15
16 int main(int argc, char *argv[])
17 {
18   // ...
19   // Indicate the semantics associated to each operation
20   mGroupAdmin::InterfaceSemantics sem;
21   sem.default_semantics_ = mGroupAccess::TOTAL_ORDER;
22   sem.operation_semantics_.length(1);
23   sem.operation_semantics_[0].name_ = "_get_balance";
24   sem.operation_semantics_[0].semantics_ = mGroupAccess::RELIABLE;
25   sem.commutative_operations_.length(1);
26   sem.commutative_operations_[0].length(2);
27   sem.commutative_operations_[0][0] = "deposit";
28   sem.commutative_operations_[0][1] = "deposit";
29   ga->join_group(server, sem);
30   // ...
31 }
```

Figure 5.10: C++ Implementation of the Account Server

Since the state of the account consists of a single floating point value, the state transfer operations are very simple. The implementations of the `get_state()` and `set_state()` operations (lines 2–14) respectively pack and unpack the current balance into and from an `any` variable.

As explained above, some operation pairs of the account's interface are not commutative. In particular, `deposit()` is not commutative with `withdraw()`, but is commutative with itself. Before joining a group, we construct a structure of type `InterfaceSemantics`, that defines the ordering guarantees required by all of the account's operations. By default, all operations require total ordering (line 21). This choice is overridden for reading the `balance` attribute (corresponding to the `_get_balance()` operation). This operation is read-only, and does not need to be

totally ordered if the clients do not need to guarantee linearizability, and do not care about receiving an outdated value. Therefore, we do not associate specific ordering guarantees for this operation (lines 23–24). Finally, we specify that the `deposit()` operation is commutative with itself (lines 25–28), allowing implementation optimizations in case of concurrent requests.

5.4 Parallel Processing with OGS

A group of objects that have the same interfaces may be used for other purposes than replication. In particular, a group of objects may be used to compute time-consuming requests in parallel. The client issues a request to the group, without having to know how many members are part of the group, and how they process the request. The group members share the work among them, compute together the result of the request, and return the reply to the client.

An example of such a distributed parallel application is the computation of the Mandelbrot set. The Mandelbrot set is a fractal structure defined in the complex plane by the following equation: $z_n = (z_{n-1})^2 + z_0$. The set itself is the area where $\lim_{n\to\infty} z_n < \infty$.

It is demonstrated that if $|z_i| > 4$, then z_n will eventually reach ∞. An approximation of the set can be computed by iterating the formula. Points where $|z_i| > 2$ are not part of the set, and the remaining points *may* be part of the set. The resulting set is traditionally displayed in a two-dimensional picture.

This computation is time-consuming: for each point the formula is iterated until $|z_i| > 2$, or a constant number of iterations have been performed. Because the adherence of each point to the set is determined only by the point's position, the computation is easy to parallelize.

5.4.1 Design

In this application, we adopt a client-server approach with the server providing the processing power while the client displays graphically the resulting set. To have the image displayed in "real-time" and to reduce the size of messages, the server transmits the data line by line, as soon as they are completed, to the client, which is updated asynchronously (Figure 5.11).

In this application, OGS is used to distribute the workload among several servers. The area of the Mandelbrot set is separated into slices, of which each is computed on a different server. The set is subdivided into n slices, where n is the number of members in the group. Each member uses its position in the current view to decide which slice to compute.

OGS gives the illusion of one single server whereas the work is actually distributed to several effective servers, making it possible to increase the parallelism degree without the knowledge of the client (by adding new servers at runtime).

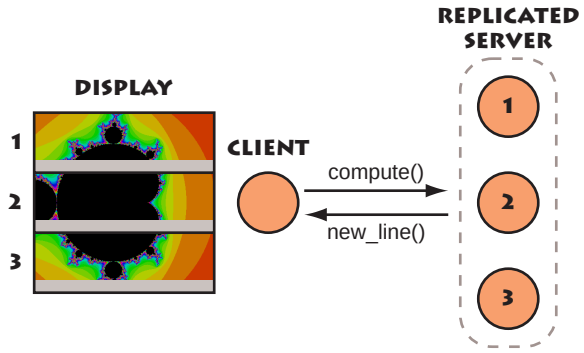

Figure 5.11: Mandelbrot Example Application

In our application, the server is implemented in C++ and the client in Java. That way, the computation benefits from C++ efficiency, while the client uses Java's graphical facilities for displaying the resulting image.

5.4.2 IDL Specification

The IDL interface of the Mandelbrot application is given in Figure 5.12. It is composed of a `Mandelbrot` module containing two interfaces: `Client` (lines 7–9) and `Server` (lines 11–16). The server's interface inherits from OGS' `Groupable` interface to become replicated.[4]

```
1  // IDL
2  module Mandelbrot
3  {
4    const long LINE_SIZE = 400;
5    typedef long Line[LINE_SIZE];
6
7    interface Client {
8      oneway void new_line(in long number, in Line data);
9    };
10
11   interface Server : mGroupAdmin::Groupable {
12     void compute(in Client client,
13                  in long top, in long left,
14                  in long height, in long width,
15                  in long iter, in long zoom);
16   };
17 };
```

Figure 5.12: IDL Interfaces of the Mandelbrot Application

[4]Notice that the client does not need to care about this inheritance relationship.

5.4.3 Client Implementation

The Mandelbrot client acts as a CORBA server and exports an object of type `Mandelbrot::Client` to the ORB. This object is responsible for receiving the Mandelbrot set computed by the servers and displaying it. The Java implementation of the client is shown in Figure 5.13.

```
1  // Java
2  public class Mandelbrot_Client extends Mandelbrot._ClientImplBase {
3    int[] matrix;
4    int top;
5    Imager img;
6
7    public Mandelbrot_Client(int top, int height)
8    {
9      matrix = new int[height*LINE_SIZE.value];
10     this.top = top;
11     img = new Imager(matrix, height, LINE_SIZE.value);
12   }
13
14   public void new_line(int number, int[] data)
15   {
16     System.out.println("Got line " + number);
17     int index = number - top;
18     System.arraycopy(data, 0, matrix, index*LINE_SIZE.value,
19                      LINE_SIZE.value);
20     img.RefreshLine(index);
21   }
22   // ...
23 }
```

Figure 5.13: C++ Implementation of the Mandelbrot Client

The client gets complete lines from the servers through the `new_line()` operation (lines 14–21), which is invoked each time a new line has been computed. The `Imager` class, whose code is not given here, is used to create and display an image corresponding to the Mandelbrot data sent by the servers. The client refreshes the display each time it gets a new line, providing graphical feedback about the ongoing computation. Since the area to compute is separated into slices, each of which is managed by a different server, the user can see these slices filling up independently at different speeds.

Notice that the `new_line()` operation of the client interface is idempotent: invoking this operation twice with the same parameters is harmless since it will simply overwrite the current line with the same data. This property allows us to handle the failure of a server in a simple way: non-failed servers can simply redo the computation of the failed server from the beginning, possibly re-sending the same lines to the client, as shown in Figure 5.14.[5]

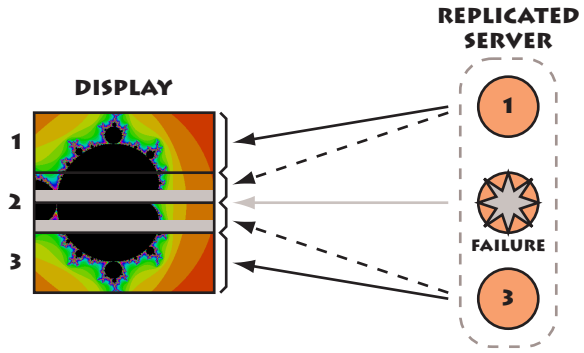

Figure 5.14: Redistribution of the Work upon Failure

```
1  // C++
2  class Mandelbrot_i : _sk_Mandelbrot::_sk_Server
3  {
4    int nb_members_;
5    int position_;
6
7  public:
8    Mandelbrot_i() : nb_members_(1), position_(0) {}
9
10   // Groupable IDL operations
11   virtual void view_change(const mGroupAccess::GroupView& new_view);
12   // ...
13
14   // Mandelbrot IDL operations
15   virtual void compute(Mandelbrot::Client_ptr client,
16                        CORBA::Long top, CORBA::Long left,
17                        CORBA::Long height, CORBA::Long width,
18                        CORBA::Long iter, CORBA::Long zoom);
19 };
```

Figure 5.15: C++ Interfaces of the Mandelbrot Server

5.4.4 Server Implementation

The C++ interface of the Mandelbrot server is shown in Figure 5.15. The most meaningful operation of the `Groupable` interface for the Mandelbrot server is the `view_change()` operation.

The information about the current view is used to decide which area of the Mandelbrot set to compute: the Mandelbrot set is divided in the same number of horizontal slices as there are members in the group, and each server computes the slice associated to its own position in the view. The view information is stored in the member's `nb_members_` and `position_` variables and is updated in the `view_change()` operation (lines 2–8), as shown in Figure 5.16. The C++ code used to compute the Mandelbrot set (lines 11–24) only computes the area allocated to the local server,

[5]Notice that this mechanism is not implemented in the code presented in this section.

and updates the client each time a new line is completed.

```
1  // C++
2  void Mandelbrot_i::view_change(const mGroupAccess::GroupView& new_view)
3  {
4    nb_members_ = new_view.composition_.length();
5    for(position_ = 0; position_ < nb_members_; position_++)
6      if(this->_is_equivalent(new_view.composition_[position_]))
7        break;
8  }
9  // ...
10
11 void Mandelbrot_i::compute(Mandelbrot::Client_ptr client,
12                            CORBA::Long top, CORBA::Long left,
13                            CORBA::Long height, CORBA::Long width,
14                            CORBA::Long iter, CORBA::Long zoom)
15 {
16   CORBA::Long start = top + position_*height/nb_members_;
17   CORBA::Long end = top + (position_+1)*height/nb_members_ - 1;
18   CORBA::Long right = left + width;
19   Mandelbrot::Line data;
20   for(int line = start; line <= end; line++) {
21     // Calculate the line (not shown)
22     client->new_line(line, data);
23   }
24 }
```

Figure 5.16: C++ Implementation of the Mandelbrot Server

5.5 Collaborative Work with OGS

Group communication is also appropriate for collaborative work, with applications communicating together using totally ordered multicasts. An example is a distributed chat application, similar to the well-known *Internet Relay Chat* (IRC) program, but without the centralized server that receives and forwards messages.[6] It allows participants all over the Internet to talk to one another in real-time. Users can join chat channels and send messages to these channels. All participants listening to the channel receive the messages. Each participant has a nickname sent along with the messages to identify the originator of the message by other users. This distributed chat application may be seen as a simple component for *collaborative authoring*.

5.5.1 Design

The distributed chat application does not have a pure client/server design. Chat channels are mapped to groups, and participants are both clients and servers of these groups. The member objects are *not* copies of a replicated object; they are distinct entities that collaborate by exchanging messages using group communication.

[6] In contrast with a solution based for instance on the CORBA event service [OMG97].

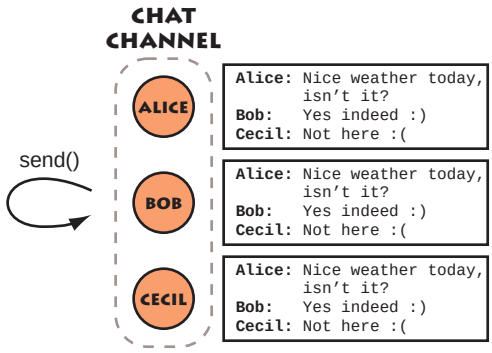

Figure 5.17: Distributed Chat Application

The general architecture of the distributed chat application is illustrated in Figure 5.17. Messages issued by a channel member are multicast to all users listening to the channel. The programming model is symmetrical: after a message has been multicast, the originator will receive and deliver its own message like all other group members.

If a new participant joins a channel, it will not get previous messages sent to the channel. Therefore, there is no need for group members to maintain a shared state.

This model does not scale well if there are hundreds of participants in the same channel. In this situation, a better architecture would be to have a small set of chat servers, and distinct clients that multicast and receive messages to and from these servers. This architecture was adopted by the Isis Distributed News Service [BCJ+90], which also provides filtering and permanent storage of messages. This model can be implemented easily with OGS.

5.5.2 IDL Specification

The architecture of this application differs from the other examples, in that there is only one type of object (the chat user), which is both client and server. Chat users send messages to their own group; the messages are asynchronously delivered to all users on the channel (including the sender) in a consistent ordering.

```
1  // IDL
2  interface ChatUser : mGroupAdmin::Groupable {
3    readonly attribute string nickname;
4
5    void post(in string sender, in string msg);
6  };
```

Figure 5.18: IDL Interfaces of the Chat Server

The IDL interface of the chat application, shown in Figure 5.18, is composed of

a single operation used to send a message to the current group, and a read-only attribute that stores the nickname of the local participant.

5.5.3 Server Implementation

```
1  // C++
2  class ChatUser_i : _sk_ChatUser
3  {
4    char *nickname_;
5
6  public:
7    ChatUser_i(char *nickname) : nickname_(nickname) {}
8
9    // Groupable IDL operations
10   virtual void view_change(const mGroupAccess::GroupView& new_view);
11   // ...
12
13   // ChatUser IDL operations
14   virtual char *nickname();
15   virtual void post(in string sender, in string msg);
16 };
```

Figure 5.19: C++ Interfaces of the Chat Server

The C++ interface of the chat server is given in Figure 5.19. In this application, a chat user object is stateless. It only receives messages, prints them to the screen, and forgets them. The most meaningful operation of the `Groupable` interface is the view change notification: each time the membership changes, the chat user displays the list of participants, as shown in Figure 5.20. Notice that we use standard CORBA invocations for obtaining the nickname of each user member of the channel.

```
1  // C++
2  void ChatUser_i::view_change(const mGroupAccess::GroupView& new_view)
3  {
4    cout << "Participants:" << endl;
5    for(int i = 0; i < new_view.composition_.length(); i++) {
6      ChatUser_var chat = ChatUser::_narrow(new_view.composition_[i]);
7      cout << i << ": " << chat->nickname() << endl;
8    }
9  }
```

Figure 5.20: C++ Implementation of the Chat Server

5.6 On-Line Software Upgrade with OGS

Upgrading a continuously available application is difficult, since it requires swapping an old version of an application with a new version, without interrupting the service

to clients. Group communication provides solutions to this kind of problems.[7] A group can be composed of a mixture of old and new versions of the application, as long as the new version is compatible with the old one, i.e., it can exhibit the same behavior as the old one. This property makes it possible to upgrade a system without stopping the service.

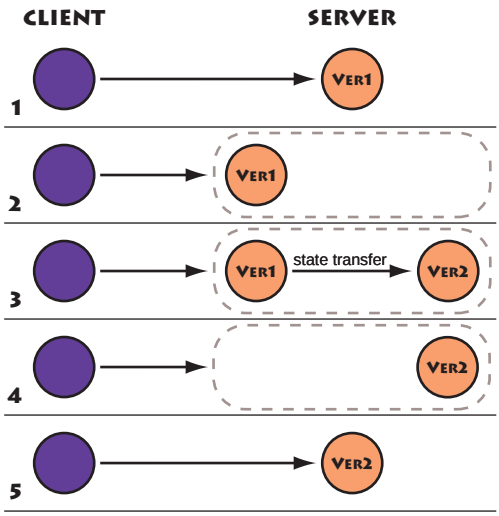

Figure 5.21: Continuous Availability using Groups for Software Upgrade

Figure 5.21 illustrates the upgrade mechanism of a continuously available system. No code is given in this section, but this mechanism may be used for any kind of application. Initially, clients use an old version of the service (1). Then, a group is formed with the old server (2), and some copies of the new version are added to the group (3). The state transfer mechanism is used to update the new copies. Finally, the remaining copies of the old version are removed (4) and the new version of the service is in operation (5).

This mechanism requires that the states of the old version and the new version are compatible (i.e., the new version of the application is able to construct its state from the state of the old version), and that the new version can behave like the old one, (i.e., it has a compatible interface). This may require to use an intermediary version, capable of behaving like both the old and the new version, to perform the upgrade. This intermediary version must be kept as long as there are clients for the old version of the application in the system.

⋆
⋆ ⋆

[7] Note that the upgrade mechanism described in this section requires messages to be totally ordered with respect to view changes and state transfers.

Summary

Many kinds of applications may benefit from OGS. OGS may be used for high availability and fault tolerance through object replication (e.g., distributed bank application), for parallel processing of time-consuming requests (e.g., computation of the Mandelbrot set), for collaborative work (e.g., collaborative edition), or for on-line software upgrades. Depending on the type of the application, groups may be composed of objects that have the same interface and perform the same task, or of objects that support different interfaces and have different behaviors.

OGS may be configured to offer several levels of reliability and transparency. It provides two execution styles: a *linkable* model and a *daemon* model. The library model is more efficient and reliable, but less transparent that the daemon model. In addition, the OGS components may be located on the same host as the user application, or may be accessed remotely. Finally, the service instantiation may be implicit, allowing us to achieve full client transparency, or explicit.

OGS makes it easy for the developer to add object groups into existing applications. If OGS is configured to provide full client transparency, the code of the client does not need to be modified or recompiled, and only few changes are necessary on the server side.

Conclusions

...but each for the joy of the working, and each, in his separate star, shall draw the thing as he sees it...

R. Kipling

Research Assessment

Distributed computing is one of the major trends in the computer industry. As systems become more distributed, they also become more complex and have to deal with new kinds of problems, such as partial crashes and link failures.

To answer the growing demand in distributed technologies, several middleware environments have emerged during the last few years. These environments however lack support for one-to-many communication primitives; such primitives greatly simplify the development of several types of applications that have requirements for high availability, fault tolerance, parallel processing, or collaborative work. Augmenting a middleware architecture by adding support for object groups will provide the developer with powerful group primitives while preserving the key features of the middleware environment.

Group Support in CORBA. CORBA is an object-oriented middleware environment that provides high-level facilities for developing distributed applications through *component integration*. Several approaches are available to add object group support to CORBA. We have classified these approaches according to three categories: the *integration* approach, the *interception* approach, and the *service* approach. We have evaluated these approaches and argued that the service approach complies best with the component-based architecture of CORBA.

A Component-Oriented Approach. In this thesis, we have proposed an open architecture for object group support in CORBA, based on a component-oriented approach. This architecture is generic and can be applied to other middleware

environments than CORBA. The *Object Group Service* (OGS) defines an object group model and specifies a set of interfaces adapted to group communication in middleware environments. OGS is decomposed into several components that conspire to provide higher-level services. This decomposition promotes modularity and reusability, and complies with the CORBA architectural model.

The OGS Architecture. The OGS components include a *group membership service*, which keeps track of the composition of object groups, a *group multicast service*, which provides delivery of messages to all group members with various guarantees, a *consensus service*, which allows several CORBA objects to solve distributed agreement problems, and a *monitoring service*, which provides distributed failure detection mechanisms. Each of these services is architecturally independent from the other ones, and most of them are reusable in areas other than group communication.

OGS provides support for dynamic group membership and for group multicast with various reliability and ordering guarantees. It defines interfaces for active and primary-backup replication. In addition, OGS proposes several execution styles and various levels of transparency. CORBA objects can easily be made groupable using interface inheritance. Client applications can communicate with object groups in the same way as they do with singleton objects.

The OGS Implementation. A prototype implementation of OGS has been developed in the context of this thesis. This implementation is available for two commercial ORBs (Orbix and VisiBroker). It relies solely on the CORBA specification, and is thus portable to any compliant ORB.

We have developed some original algorithms for implementing group communication in OGS. Group membership and total order are implemented using a consensus algorithm. Actively replicated objects can optionally use an efficient sequencer-based algorithm. Primary-backup replication is based on a consensus algorithm with deferred initial values.

Performance measurements show that the cost of OGS is not excessive, when compared to direct invocations through the ORB. The overhead of OGS can be classified according to three categories: the cost of *plurality* corresponds to the overhead induced by the addition of members to the group; the cost of *behavior* designates the individual performances of the various communication protocols that correspond to different group behaviors; the cost of *typing* corresponds to the overhead of using typed over untyped communication.

Standard Group Support for CORBA

In April 1998, the OMG issued a *Request For Proposals* (RFP) entitled: *Fault Tolerant CORBA Using Entity Redundancy* [OMG98b]. The goal of this RFP is to address the need for standard CORBA mechanisms supporting fault tolerant applications. Such mechanisms include redundant copies, failure masking, and failure

recovery.

The RFP does not specify the unit of redundancy, but uses the term "entity". It presents the concepts — developed in this dissertation — of *entity groups*, with dynamic group membership, and that operate in a number of ways including active and primary-backup replication. Among the mandatory requirements of the RFP are specification of the unit of redundancy; definition of interfaces for passive entity groups; support for state synchronization; support for dynamic group management; interfaces that allow entity redundancy to be transparent to clients; ability of entity groups to determine failed members; interfaces for recovery control. In addition, the RFP defines some optional requirements, such as support for active entity groups; interfaces for using failure detection and notification mechanisms; support for suppressing multiple responses from an entity group to a single request.

OGS vs. Fault Tolerant CORBA. While the object group service presented in this dissertation does not address *all* the issues raised by the RFP, we strongly believe that the contribution of this work and the lessons learned from our experiences can be useful for a proposal to this RFP. As a matter of fact, OGS offers solutions to most of the requirements specified in the RFP, and a prototype implementation is available as a proof of concept.

One should notice that the approach developed in this research does not extend or modify the CORBA specification at all. This is a design choice that we made to ensure the portability of our implementation and the compliance with existing ORB implementations. Actual responses to an RFP may specify required changes or extensions to the existing CORBA standard. Such changes may allow better integration of a service with the ORB and other services.

New CORBA Technologies

As mentioned in Chapter 4, the current CORBA specifications have several limitations concerning code portability and *Quality of Service* (QoS). Some of these limitations should be alleviated by the proposed messaging service [BEI+98] and the new CORBA 2.2 specification [OMG98a].

OMG's Messaging Service. The OMG is currently specifying a new messaging service, which is not yet finalized at the time of this writing. We base the following description on the joint revised submission to the messaging service [BEI+98].

The messaging specification will provide two truly asynchronous method invocation models, based on *callback* and *polling*. In the first model, a callback object is registered at the time of the invocation. When the reply is available, the callback object is invoked with the data of the reply. In the polling model, the invocation returns an object which can be queried at any time to obtain the status of the outstanding request. It will be possible to issue asynchronous invocations using static interfaces, thus maintaining strong typing.

Whereas current CORBA specifications do not address the kinds of QoS features associated with messaging systems, the messaging specification will provide several kinds of QoS such as delivery characteristics (lifetime of request and reply, reliability, scope of synchronization with target) and server-side queue management or ordering (temporal based, priority based). The messaging QoS can be used to request a specific protocol as well as to guide the implicit protocol selection by the ORB. QoS can be independently specified at different levels: ORB, thread, object, or object adapter.

The messaging specification will thus circumvent the shortcomings resulting from the lack of a truly asynchronous method invocation model in CORBA. Migration of OGS code to this service will be easy, thanks to the definition of a messaging service in the OGS architecture.

The CORBA Portable Object Adapter. The new CORBA 2.2 specification [OMG98a] defines a new *Portable Object Adapter (POA)* meant to replace the much criticized BOA. As suggested by its name, one of the major design goal of this new adaptor is to allow programmers to construct object implementations that are portable between different ORBs. The CORBA 2.2 specification also specifies how the POA maps to programming languages. With the POA and the new mappings, it will be possible to develop fully portable servers.

The new POA standardizes threading models. The POA supports two models of threading when used in conjunction with multi-threaded ORB implementations: ORB controlled and single thread behavior. The two models can be used together or independently. The threading model associated with a POA is indicated upon POA creation using a `ThreadPolicy` object. The ORB interface also defines some new thread-related operations in the ORB interface. Although this threading support is minimal, OGS will be able to use basic thread mechanisms in a portable way.

The POA specification now states that objects can be co-located (location transparency), and that invocations to local objects will be mediated by the ORB through the POA the same way as remote invocations. This will allow us to develop portable components that have locality constraints, and to use CORBA dynamic typing facilities (DSI and DII) with co-located objects.[8]

The C++ mapping of the new CORBA specification mandates that exceptions can be inserted in and extracted from `any` variables, thus allowing OGS to return back exceptions to the client when using typed communication.

The CORBA 2.2 specification also standardizes the following features that are relevant to future implementations of OGS, although they are not directly related to problems presented in this dissertation. The new POA adds support for objects with persistent identities. The POA is designed to allow programmers to build object implementations for objects whose lifetime (from the perspective of a client holding a reference for such an object) span multiple server lifetimes. This will be very useful for providing recovery of crashed servers.

[8] OGS cannot use the DSI with co-located objects because this model is not supported by Orbix and VisiBroker.

The POA also allows the association of a single DSI servant[9] with many CORBA objects. This is especially interesting for applications like OGS, since a single service object will be able to mediate requests for several replicated servers that support different interfaces.

Another addition of the CORBA 2.2 specification is dynamic management of `any` values. If an `any` is passed to a program that does not have any static information for the type of the `any`, the object receiving the `any` does not have a portable method for using it. The new specification enables traversal of the data value associated with an `any` at runtime and extraction of its primitive constituents, as well as the construction of an `any` at runtime without having static knowledge of its type. These facilities are especially useful for writing powerful generic services, and OGS will directly benefit from it (e.g., for filtering invocations, or for returning exceptions to clients).

Finally, the addition of `Interceptor` objects makes development of services much easier. An interceptor is an object interposed in the invocation and response paths between a client and a target object. Interceptors may be of two kinds: request-level and message-level, and several interceptors may be chained. Interceptors permit services that stand close to the ORB to be cleanly separated. By using interceptors OGS will be able to coexist better with other ORB services, such as the security service.

OGS for Other Environments

Although the OGS implementation strongly depends on CORBA, the OGS system model and architecture can be applied to other middleware environments. In particular, Microsoft's *Distributed Component Object Model* (DCOM) [Ses97] has many similarities with CORBA [CHY+98]. Both are middleware environments that allow distributed components to communicate with each other using remote method invocations. DCOM uses a purely declarative interface definition language comparable to OMG's IDL which provides a clean separation of interfaces and implementations.

At first glance, applying the concepts presented in the thesis to DCOM would not present major problems. In particular, DCOM provides a data type similar to the CORBA `any` type[10] and dynamic invocation mechanisms that could be used to implement both untyped and typed group communication. However, many implementation details will certainly differ when adapting the OGS architecture to DCOM.

[9] A *servant* corresponds to an object implementation.

[10] Actually, the `Variant` type of DCOM can only contain simple types, interfaces, and arrays of these types.

Open Issues and Future Work

OGS defines a complete framework for object group support in CORBA and a usable prototype has been realized. However, several issues still remain open. Some ideas of OGS extensions and applicability domains are outlined below.

OGS Extensions. As already mentioned, we decided in this thesis to rely only on the CORBA specification, and to use off-the-shelf ORB implementations. This design choice has some shortcomings, since better integration of OGS would be possible with some additional support from the ORB.

For instance, similarly to request filtering, OGS does not provide a clean solution to the problem of group members returning object references. Consider a replicated bank application that creates accounts for its clients. The accounts are created in the address space of the bank servers. Each of these servers returns a reference to the newly created account object, but these references are different (as specified by the CORBA identity model). In this situation, the correct behavior would be to gather all the account objects in a group and to return a group reference to the client. This group would not be named and its members would be co-located with their respective creator; we call this type of groups *anonymous shadow groups.*[11] Unfortunately, this mechanism cannot be implemented without some support from the ORB.

New Components in OGS. OGS is an open environment, and new components may be added without requiring modifications to existing ones. Some interesting additions to the OGS environment could be:

- *Replication service:* this component would define high-level interfaces dedicated to replication, without presenting object groups to the application. It would provide configuration tools to specify in a simple way the replication policy and the redundancy degree of replicated objects.
- *Recovery service:* this component would automate recovery of failed objects.

Fault Tolerant Services using OGS. OGS could be used to implement fault tolerant and highly available CORBA services [OMG97]. For instance, replicated event channels or a fault tolerant naming service could be implemented using OGS.

Interactions with Other Services. One could study the interactions of OGS and existing services [OMG97], such as transactions, persistence, security, life cycle, or notification. For instance, the life cycle service could be used for transferring the state of group members, or to dynamically augment the redundancy level of an object group. Another example is the support of replicated transactions; the interactions

[11] A shadow group is a group that maps exactly to another group.

between OGS and OMG's transaction service are complex, and supporting replicated transactions could necessitate modifications to both services.

Protocol Extensions. New protocols can be added to OGS without impact on the existing architecture. These protocols can use the OGS infrastructure (e.g., failure detection, consensus), making their development simple and fast. Therefore, OGS offers a framework that can be easily augmented with new protocols, and that can be used as a test environment for distributed algorithms as well.

One could also extend the internal architecture of OGS and develop an extensible protocol framework, that would allow us to construct new protocols by composing them. An approach similar to the Horus protocol composition framework [vRBF+95] would be adequate in the OGS architecture.

OGS for Java. A prototype implementation of OGS has been realized in the Java programming language. Besides binary compatibility, a major advantage of this approach is that clients can download the OGS runtime through the Internet; they do not need to have a local daemon or library implementation of OGS. It would be interesting to compare the performances of this Java prototype with the ones of the C++ implementation, and to validate OGS interoperability and language heterogeneity with object groups composed of a mixture of Java and C++ objects.

Bibliography

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 76–84, July 1992.

[AL81] T. Anderson and T. Lee. *Fault-tolerance Principles and Practice.* Prentice Hall, 1981.

[AMMS+95] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[AMMSB98] D.A. Agarwal, L.E. Moser, P.M. Melliar-Smith, and R.K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[Bak97] S. Baker. *CORBA Distributed Objects Using Orbix.* Addison-Wesley, 1997.

[BC91] K. Birman and R. Cooper. The Isis project: Real experience with a fault tolerant programming system. *ACM Operating Systems Review, SIGOPS*, 25(2):103–107, 1991.

[BCJ+90] K. Birman, R. Cooper, T.A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual.* Dept. of Computer Science, Cornell University, September 1990.

[BEI+98] BEA Systems, Expersoft, Inprise, IBM, ICL, IONA, Nortel, Novell, Oracle, PeerLogic, and TIBCO. *Asynchronous Messaging, Joint Revised Submission (orbos/98-05-05).* OMG, May 1998.

[BGL98] J.-P. Briot, R. Guerraoui, and K.-P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, September 1998.

[BI93] A.P. Black and M.P. Immel. Encapsulating plurality. In Oscar M. Niestraz, editor, *ECOOP'93-Object-Oriented Programming, 7th European Conference*, number 707 in Lecture Notes in Computer Science, pages 56–79. Springer-Verlag, 1993.

[Bir93] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[Bir96] K. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Co., 1996.

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BMD93] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in distributed computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[BMST93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. *Distributed Systems*, chapter 8: The Primary-Backup Approach, pages 199–216. Addison-Wesley, 2nd edition, 1993.

[BN84] A.D. Birrel and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Boo94] G. Booch. *Object Oriented Design With Applications, second edition.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[BV93] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1993.

[BvR94] K. Birman and R. van Renessee. RPC considered inadequate. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 68–78. IEEE Computer Society Press, 1994.

[CD88] G. Coulouris and J. Dollimore. *Distributed Systems.* Addison-Wesley, 1988.

[CHT96] T.D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[CHY+98] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, J.C. Shih, C.-Y. Wang, and Y.M. Wang. DCOM and CORBA side by side, step by step, and layer by layer. *C++ Report*, 10(1), Jan 1998.

[Coo85] E.C. Cooper. Replicated distributed programs. *ACM Operating Systems Review*, 19(5):63–78, December 1985.

[Coo86] E.C. Cooper. Replicated procedure call. *ACM Operating Systems Review*, 20(1):44–56, January 1986.

[CT96] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[CZ85] D.R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DFGG97] X. Défago, P. Felber, B. Garbinato, and R. Guerraoui. Reliability with CORBA event channels. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 237–240, Portland, Oregon, July 1997. USENIX Association.

[DSC92] A. Dave, M. Sefika, and R.H. Campbell. Proxies, application interfaces, and distributed systems. *Proceedings of the International Workshop on Object Orientation in Operating Systems*, 1992.

[DSS98a] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS-17)*, West Lafayette, Indiana, USA, October 1998.

[DSS98b] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. Technical Report 98/277, Ecole Polytechnique Fédérale de Lausanne, May 1998.

[ES86] P. Ezhilchevan and S. Shrivastava. A characterization of faults in systems. In *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database systems*, January 1986.

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, January 1990. Reprinted with corrections January, 1994.

[FG95] P. Felber and R. Guerraoui. Group programming: an object-oriented approach. In *Technology of Object-Oriented Languages and Systems (TOOLS 16)*, pages 263–271, Paris, March 1995. Prentice-Hall.

[FGG96] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 150–159, October 1996.

[FGG97] P. Felber, B. Garbinato, and R. Guerraoui. *Special Issues in Object-Oriented Programming*, chapter Towards Reliable CORBA: Integration vs. Service Approach, pages 199–205. dpunkt-Verlag, 1997.

[FGS97] P. Felber, R. Guerraoui, and A. Schiper. Replicating objects using the CORBA event service. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'97)*, pages 14–19, October 1997.

[FGS98a] P. Felber, R. Guerraoui, and A. Schiper. Evaluating CORBA portability: The case of an object group service. In *Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop (EDOC'98)*, San Diego (California), November 1998.

[FGS98b] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[FLP85] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. In *Journal of the ACM*, volume 32, pages 217–246, 1985.

[FNPW95] J.-C. Fabre, V. Nicomette, T. Pérennou, and R. Stroudand Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 489–498, Pasadena (California), June 1995.

[Gar98] B. Garbinato. *Protocol Objects & Patterns for Structuring Reliable Distributed Systems*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1801.

[GFG96] B. Garbinato, P. Felber, and R. Guerraoui. Protocol classes for designing reliable distributed environments. In *Proceedings of the 10th European Conference on Object Oriented Programming (ECOOP), Linz (Austria)*, number 1098 in Lecture Notes in Computer Science, pages 316–343. Springer-Verlag, July 1996.

[GFG97a] B. Garbinato, P. Felber, and R. Guerraoui. Modeling protocols as objects for structuring reliable distributed systems. In *Communication Networks and Distributed Systems Modeling and Simulation Conference*, January 1997.

[GFG97b] B. Garbinato, P. Felber, and R. Guerraoui. *Special Issues in Object-Oriented Programming*, chapter Right Abstractions on Adequate Frameworks for Building Adaptable Distributed Applications, pages 24–28. dpunkt-Verlag, 1997.

[GFGM98] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System support for object groups. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, 1998.

[GGM93] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Programming fault-tolerant applications using two orthogonal object levels. In *Proceedings of the 8th International Symposium on Computer and Information Sciences (ISCIS-8)*, Istambul (Turkey), November 1993.

[GGM94] B. Garbinato, R. Guerraoui, and K.R. Mazouni. Distributed programming in GARF. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object Based Distributed Programming*, number 901 in Lecture Notes in Computer Science, pages 225–239. Springer Verlag, 1994.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, August 1996.

[Gou92] Y. Gourhant. An object-oriented approach for replication management. In *Proceedings of the 2nd Workshop on the Management of Replicated Data*, pages 74–77. IEEE Computer Society Press, November 1992.

[GR83] A.J. Goldberg and A.D. Robson. *SMALLTALK-80: The Language and its Implementation*. Addison Wesley, 1983.

[GS95] R. Guerraoui and A. Schiper. Transaction model vs. virtual synchrony model: Bridging the gap. In *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 121–132. Springer Verlag, 1995.

[GS96] R. Guerraoui and A. Schiper. Consensus service: a modular approach for building agreement protocols in distributed systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 168–177, June 1996.

[GS97] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'97)*, pages 183–188, October 1997.

[GS98] A. Gokhale and D. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *IEEE Transactions on Computers*, 47(4):391–413, April 1998.

[Hay98] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University Computer Science Department, January 1998.

[Hil96] M.A. Hiltunen. *Configurable Fault-Tolerant Distributed Services*. PhD thesis, University of Arizona, June 1996.

[HJE95] H. Hüni, R. Johnson, and R. Engel. A framework for network protocol software. In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*. ACM Press, 1995. Special Issue of Sigplan Notices.

[HT93] V. Hadzilacos and S. Toueg. *Distributed Systems*, chapter 5: Fault-Tolerant Broadcasts and Related Problems, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[HW90] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[II94] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[Inc93] Isis Distributed Systems, Inc. Object groups: a response to the ORB 2.0 RFI. April 1993.

[ION97] IONA. *Orbix 2.2 Programming Guide*. IONA Technologies Ltd., Mar 1997.

[JF88] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[Jul93] E. Jul. Separation of distribution and objects. In R. Guerraoui and O. Niestrasz amd M. Riveill, editors, *Object-Based Distributed Programming, ECOOP'93 Workshop Proceedings*, number 791 in Lecture Notes in Computer Science, pages 46–54. Springer-Verlag, July 1993.

[Kon93] D. Konstantas. Object oriented interoperability. In Oscar M. Niestraz, editor, *ECOOP'93-Object-Oriented Programming, 7th European Conference*, number 707 in Lecture Notes in Computer Science, pages 81–102. Springer-Verlag, 1993.

[KR77] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1977.

[KT91] F. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[Lap91] J.-C. Laprie. Dependability concepts. In D. Powell, editor, *Delta-4 A Generic Architecture for Dependable Distributed Computing*, pages 43–69. Springer-Verlag, 1991.

[LCY97] D. Liang, S.C. Chou, and S.M. Yuan. Adding fault-tolerant object services to CORBA. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[Lea97] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[LS94] M.C. Little and S.K. Shrivastava. Object replication in Arjuna. Technical report, University of Newcastle, 1994.

[LSP82] L. Lamport, R.E. Shostak, and M.C. Pease. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[Maf95] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich, February 1995.

[Maf96] S. Maffeis. A fault-tolerant CORBA name server. In *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems*, pages 188–197, October 1996.

[Mal96] C.P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, September 1996. Number 1557.

[Maz96] K.R. Mazouni. *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1996. Number 1578.

[MGG95] K.R. Mazouni, B. Garbinato, and R. Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proceedings of the 4th IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Lund (Sweden), August 1995.

[MMSA94] L.E. Moser, P.M. Melliar-Smith, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[MMSA+96] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[MMSN98] L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[MSMA90] P.M. Melliar-Smith, L.E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[NBF96] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly, 1996.

[NMMS97a] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Exploiting the internet inter-ORB protocol to provide CORBA with fault tolerance. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 81–90, Portland, Oregon, July 1997. USENIX Association.

[NMMS97b] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Replica consistency of CORBA objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, September 1997.

[OMG97] OMG. *CORBAservices: Common Object Services Specification.* OMG, 1997.

[OMG98a] OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, February 1998.

[OMG98b] OMG. *Fault tolerant CORBA Using Entity Redundancy, Request For Proposal.* OMG, Apr 1998.

[OPR96] R. Otte, P. Patrick, and M. Roy. *Understanding CORBA.* Prentice-Hall, 1996.

[Pow94] D. Powell. Distributed fault tolerance: Lessons from Delta-4. *IEEE Micro*, pages 37–47, February 1994.

[PV91] D. Powell and P. Veríssimo. Distributed fault-tolerance. In D. Powell, editor, *Delta-4 A Generic Architecture for Dependable Distributed Computing*, pages 89–124. Springer-Verlag, 1991.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Rei95] M.K. Reiter. The Rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science)*, 938:99–110, 1995.

[Sch93a] F. Schneider. *Distributed Systems*, chapter 7: Replication Management using the State-Machine Approach, pages 169–197. Addison-Wesley, 2nd edition, 1993.

[Sch93b] F. Schneider. *Distributed Systems*, chapter 2: What Good are Models and What Models are Good?, pages 17–26. Addison-Wesley, 2nd edition, 1993.

[Sch94] D.C. Schmidt. ASX: an object-oriented framework for developing distributed applications. In *Proceedings of the 6th USENIX C++ Conference*, 1994.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[Ses97] R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* John Wiley & Sons, 1997.

[Sha86] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society Press, May 1986.

[SM94] S.K. Shrivastava and D.L. McCue. Structuring fault-tolerant object systems for modularity in a distributed environment. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):421–432, April 1994.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[SS93] A. Schiper and A. Sandoz. Understanding the power of the virtually-synchronous model. In *Proceedings of the 5th European Workshop on Dependable Computing*, 1993.

[Str97] B. Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[Sun90] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Uma93] A. Umar. *Distributed Computing.* Prentice-Hall, 1993.

[VB98] A. Vaysburd and K. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.

[Vis98a] Visigenic. *VisiBroker for C++ 3.2 Programmer's Guide.* Visigenic Software, Inc., Mar 1998.

[Vis98b] Visigenic. *VisiBroker for Java 3.2 Programmer's Guide.* Visigenic Software, Inc., Feb 1998.

[vN56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McMarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[vRBF+95] R. van Renesse, K.P. Birman, R. Friedman, M. Hayden, and D.A. Karr. A framework for protocol composition in Horus. In *Proceedings of Principles of Distributed Computing*, August 1995.

[vRBM96] R. van Renesse, K.P. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4), April 1996.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

Curriculum Vitae

Pascal Felber was born on March 16, 1971, in Lausanne, Switzerland. He attended to Gymnase de la Cité, Lausanne, where in 1989 he obtained a high school degree in classical studies (Latin/English). In 1994, he received his M.S. in Computer Science from the Swiss Federal Institute of Technology, Lausanne (EPFL). Since then, he has been working as a research and teaching assistant, and as a Ph.D. student in the Operating Systems Laboratory (LSE) of the EPFL.

During his stay in the LSE, he has been involved in several European projects: Broadcast (project 6360), OpenDREAMS (project 20843), and OpenDREAMS II (project 25262). His research focused on reliable, object-based distributed systems. He also worked as system administrator of the LSE from 1995 to 1997.